Microcross i.MX GNU X-Tools™

Professional Embedded Development Toolsuite

User Guide

Version 3.40d

© 2005-2006 Microcross, Inc.
Licensed Materials

Open-Source, Ready-to-Run ™

Contents

ISBN: N/A

Microcross, Inc.
104 Borders Way, Suite 100
Warner Robins, GA 31088
USA

Copyright © 2005-2006 Microcross, Inc.
All rights reserved.

Trademarks
GNU X-Tools™ and Microcross™ are all trademarks of Microcross, Inc.. This documentation contains copyright materials and has been prepared by Microcross Technical Publications; contact the Microcross Technical Publications staff for more information: **support@microcross.com**.

ARM™, Thumb™, and ARM Powered™ are registered trademarks of ARM Limited. All other brand and product names, services names, trademarks and copyrights are the property of their respective owners.

Disclaimer
Microcross, Incorporated makes no representations or warranties with respect to the contents or use of this user guide, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Microcross, Incorporated reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes. Microcross, Incorporated makes no representations or warranties with respect to any Microcross software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose.

i.MX GNU X-Tools™ User Guide
Supports Product Part Number: 336-75397, January 2006

CONTENTS

SECTION 1. INTRODUCTION 7

1.1 What is i.MX GNU X-Tools™? 7

1.2 Acknowledgements 7

1.3 License Agreement 7

1.4 Toolsuite Components 8
1.4.1 Naming Conventions Used in GNU X-Tools 8
1.4.2 Targets 8

1.5 System Requirements and Prerequisites 9
1.5.1 System Requirements (Windows Host Version) 9
1.5.2 System Requirements (Linux Host Version) 9

1.6 Compatible RTOS/OS Software 10

1.7 Compatible JTAG/BDM Hardware 10

SECTION 2. INSTALL AND SETUP PROCEDURES 11

2.1 Install i.MX GNU X-Tools on Windows Host 11

2.2 Uninstall i.MX GNU X-Tools on Windows Host 12

2.3 Install i.MX GNU X-Tools on Linux Host 12

2.4 Uninstall i.MX GNU X-Tools on Linux Host 14

2.5 Setup Test 14

2.6 Troubleshooting Guide for Installation Problems 14
2.6.1 Windows / Cygwin Related Problems 14
2.6.2 Make.exe Related Problems 15
2.6.3 Remove Carriage Returns in Source and Make Files 15
2.6.4 'xtools install' Command Fails on Linux Host 16

SECTION 3. QUICK START GUIDE 17

3.1 i.MX LiteKit Setup 17

3.2 Build, Load, and Run Example Applications on MicroMonitor 19

3.3 Build, Load, and Run GX-Linux and Example Applications 20

3.4 Debug with usbDemon Example 23
3.4.1 Example Program 23
3.4.2 OCDRemote Monitor Commands 28
3.4.3 Some Useful GDB Console Commands 28

3.5 Visual X-Tools Example 29

3.6 LCD Tests 33

3.7 Rebuild MicroMonitor and Reflash Board 33
3.7.1 Normal Reflash Procedure 33
3.7.2 Dissaster Recovery Procedure 34

3.8 MicroMonitor Familiarization and Tips 37
3.8.1 Manual TFTP Up-Load Procedure 38
3.8.2 Other Useful MicroMonitor Commands 39

SECTION 4. HOW TO USE I.MX GNU X-TOOLS 40

4.1 Introduction 40

4.2 Using the i.MX GNU X-Tools Command Line Tool (xtools) 40

4.3 Invoking i.MX GNU X-Tools 41

4.4 Using the i.MX GNU X-Tools Toolsuite 42
4.4.1 Simple Example 42
4.4.2 i.MX GNU X-Tools Toolsuite Description 43
4.4.3 Control Program (GCC) 44

4.5 Controlling the Tools Using the GCC 47
4.5.1 GCC Options Commonly Used 48

4.6 Controlling Linker from GCC 50

4.7 Compilation Options 51
4.7.1 Displaying compiler behavior 51
4.7.2 C Language Options 51
4.7.3 Preprocessor options 52
4.7.4 Options to Specify Libraries, Paths and Startup Files 52
4.7.5 Debugging and Profiling Options 54
4.7.6 Optimization 55
4.7.7 Passing Options to the Assembler or Linker 56

4.8 Using the GNU Assembler 56

4.9 Using the Linker 59
4.9.1 Invoking ld 59
4.9.2 Linker Options 60
4.9.3 Linker Scripts 61
4.9.4 Link-Order Optimization 62

4.9.5 The C Runtime (crt0) *62*

4.10 Object Translation (ELF to Binary, SREC, etc.) 62

4.11 Creating/Updating Libraries 63

4.12 GNU Libraries 64

4.13 Instruction Set Simulator (ISS) Options 64

SECTION 5. HOW TO USE COMMAND LINE GDB 65

5.1 Summary of GDB, the GNU Debugger 65
5.1.1 GDB as Free Software *65*
5.1.2 Requirements of GDB *65*
5.1.3 Startup GDB *65*
5.1.4 Startup of GDB with DDD *66*

5.2 GDB Comprehensive Quick Reference 67
Table 5.1. Essential Commands *67*
Table 5.2. Starting GDB *68*
Table 5.3. Stopping GDB *68*
Table 5.4. Getting Help *68*
Table 5.5. Executing your Program *68*
Table 5.6. Shell Commands *68*
Table 5.7. Breakpoints and Watchpoints *68*
Table 5.8. Program Stack *69*
Table 5.9. Execution Control *69*
Table 5.10. Display *70*
Table 5.11. Automatic Display *70*
Table 5.12. Expressions *70*
Table 5.13. Symbol Table *71*
Table 5.14. GDB Scripts *71*
Table 5.15. Signals *71*
Table 5.16. Debugging Targets *71*
Table 5.17. Controlling GDB *72*
Table 5.18. Working Files *72*
Table 5.19. Source Files *73*

SECTION 6. HOW TO USE VISUAL GDB DEBUGGER 74

6.1 Using Visual GDB Debugger 74

6.2 Visual GDB, An Alternative Interface to Command Line 74

6.3 Using the Mouse in the Source Window 79

6.4 Left column functionality for the Source Window 80

6.5 Right Column Functionality for the Source Window 81

6.6 Source Window Menus and Display Features 81

6.7 Below the Horizontal Scroll Bar of the Source Window....82

6.8 Using the Stack Window 84

6.9 Using the Registers Window 84

6.10 Using the Memory Window 85

6.11 Using the Watch Expressions Window....86

6.12 Using the Local Variables Window....88

6.13 Using the Breakpoints Window 90

6.14 Using the Console Window 92

6.15 Using the Function Browser Window 93

6.16 Using the Processes Window for Threads 94

6.17 Using the Help Window....94

6.18 Examples of Debugging with Visual GDB 95
6.18.1 Selecting and Examining a Source File....95
6.18.2 Setting Breakpoints on Multiple Threads....99

6.19 Visual GDB with JTAG/BDM Debug Agents 100
6.19.1 Abatron BDI2000 Setup and Debug with Visual GDB....101
5.19.2 Macraigor mpDemon Setup and Debug with Visual GDB....103

SECTION 7. INTRODUCTION TO CYGWIN....112

7.1 Introducing Cygwin....112

7.2 Cygwin Key Features 112

7.3 Cygwin Components 112
7.3.1 Cygwin Package List – tools, utilities, and programs that make up Cygwin....112
7.3.2 User Information – How to use the tools, utilities, and programs in Cygwin 113

APPENDIX 1. ARM TOOLSUITE....115

BIBLIOGRAPHY 125

GLOSSARY OF EMBEDDED SYSTEM TERMINOLOGY....127

INDEX....132

Section 1. Introduction

1.1 What is i.MX GNU X-Tools™?

The i.MX GNU X-Tools™ product is a cross-tools development toolsuite consisting of C/C++ compilers, assembler, archiver, binary utilities, C/C++ libraries, and debugger. The GNU cross-tools collection is made up of open source cross-compilation and debugging tools designed to support professional embedded systems development for ARM9™ / ARM11™ target processors on the Linux and Windows host platforms. Microcross builds, debugs, patches, documents, and brands their product for the Freescale Semiconductor i.MX processors.

Background: The GNU tools are the evolutionary embodiment of the Free Software Foundation (FSF) GNU Compiler Collection (GCC) as envisioned by Richard Stallman of the FSF in the mid-to-late eighties. Through the collective efforts of hundreds of volunteers and organizations (see Section 1.2), the GNU Tools has grown to be a mature and robust suite of tools that is readily available in source form. The source code is freely distributable under the license terms of the General Public License (GPL) from the FSF (see www.gnu.org); however, building a toolsuite is a complex and time-consuming task. Microcross' contribution to value-add is to build, test, package, document, distribute and support the GNU toolsuites in ready-to-run form, making them accessible at low cost to both casual and professional users. The The i.MX GNU X-Tools provides all of the components an embedded developer needs in order to evaluate and use the GNU cross-toolsuites. GDBTK, also branded by Microcross as Visual GDB™ (a.k.a. Red Hat Insight debugger) is seamlessly integrated as the GUI debugger for the i.MX GNU X-Tools toolsuite.

1.2 Acknowledgements

The GNU Tools and utilities represent the essence of the free and open source movement; in fact all who participate in using and improving them are benefactors as well as contributors. Microcross is certainly no exception, and as such we gratefully acknowledge the following individuals and organizations for their contributions to the open source universe:

- Richard Stallman, founder of the GCC and the Free Software Foundation.
- The Free Software Foundation for their role in focusing and managing open source development.
- Cygnus Solutions and now Red Hat for their leadership and contributions toward the GNU Tools as a viable cross-development platform.
- Linus Torvalds for creating the Linux kernel and founding the crusade to level the OS playing field.
- Many thousands of individual contributors whose names can be found buried in the source files that they spent long nights developing, debugging and perfecting.
- Additional acknowledgements are noted in the 'contributors' file of each subdirectory of the source tree.

1.3 License Agreement

The i.MX GNU X-Tools is distributed under the license terms of the General Public License (GPL). You may read this license at www.gnu.org or on the distribution CD for i.MX GNU X-Tools. The following exceptions apply:

- The C and Math Libraries (Newlib) are distributed under an unrestricted license so that linked library binaries may be distributed as part of user developed applications without restrictions from the GPL.
- Libstdc++, the C++ class and template libraries, are distributed under the FSF GNU Lesser General Public License (LGPL).
- The i.MX GNU X-Tools and Cygwin Windows installers are licensed software material from Microcross.
- The i.MX GNU X-Tools and Cygwin CDs may not be reproduced without prior written consent from Microcross.
- The MicroMonitor source code (boot ROM for Cogent i.MX single board computers) is available under non-restrictive terms of the Lucent Public License.

1.4 Toolsuite Components

1.4.1 Naming Conventions Used in GNU X-Tools

Because the GNU cross-tools have a one-to-one correspondence in function and naming across targets, all of the GNU cross-tools are named using a target specific prefix. This target specific prefix, also called the **target alias**, is pre-pended to the name of each program binary name so that ambiguity with the host platform GNU Tools and other target toolsuites is prevented. The standard toolsuite components for each processor supported are listed in Table 1.1. Other components of the product are listed in Table 1.2. A feature of the i.MX GNU X-Tools 3.40 command shell environment is that it provides alias names for each target specific program so that you may refer to it by its generic name when running within to the tool shell. For example, after opening a GNU X-Tools Shell using the command 'xtools arm-elf', you can refer to the 'arm-elf-gcc' compiler simply as GCC. The alias remains in effect until you exit the shell at which time the command GCC reverts to the native compiler.

Table 1.1. i.MX GNU X-Tools Standard Toolsuite Components

Tool Name	Description
<target-alias>-gcc	GNU Compiler Collection (GCC) with control to the C compiler
<target-alias>-g++	C++ compiler (g++)
<target-alias>-as	GNU assembler (as)
<target-alias>-ld	GNU linker (ld)
<target-alias>-run	Instruction Set Simulator (ISS)
<target-alias>-addr2line	Converts addresses into file names and line numbers
<target-alias>-ar	Creates, modifies & extracts from object code archives
<target-alias>-gcov	Test coverage program to get basic performance statistics
<target-alias>-nm	Lists symbols from object files
<target-alias>-objcopy	Copies and translates object files
<target-alias>-objdump	Displays information from object files
<target-alias>-ranlib	Generates index to archive contents
<target-alias>-readelf	Displays information about ELF format object files
<target-alias>-size	Lists file section sizes and total sizes
<target-alias>-strings	Lists printable strings from files
<target-alias>-strip	Strips debug symbols from binaries
<target-alias>-gdb	Command Line GDB debugger
<target-alias>-gdbtk	Visual GDB debugger (a.k.a. Insight)
libc.a and libm.a	Unrestricted Newlib C and Math Libraries
libstdc++.a	GNU Standard C++ and Template Library

Table 1.2. Sources for Other Components

Tool Name	Description
bench++	A comprehensive set of C++ benchmarks that may be used to verify correct operation of the toolsuites and serve as the basis of user programs. Bench++ is completely described in the user notes for Bench++ in the /docs directory on the i.MX GNU X-Tools CD-ROM.
Newlib	Newlib i.MX C source tree. This library does not have restrictions based on the General Public License (GPL). This library is completely unrestricted. Contact Microcross for more information.

1.4.2 Targets

The array of target processors supported by i.MX GNU X-Tools is based on the current Freescale Semiconductor i.MX targets including MXL, MX1, MX21 and MX31 variants. The objective of i.MX GNU X-Tools Release 3.40 is to

provide a baselined toolsuite for each target platform and each host. From this baseline, new hosts, targets, and target enhancements will be added in future releases.

1.5 System Requirements and Prerequisites

The i.MX GNU X-Tools provides a GNU Unix / Linux like development environment on Windows and Linux hosts. Microcross presumes that the user has a working knowledge of Unix / Linux command line utilities and software tools. If not, the user should acquire training materials to get acquainted with Linux / Cygwin. This user guide is ideally suited for developers who have some prior experience either with GNU as a cross-compiler toolsuite or as a native development system. If you need additional familiarization with the Unix / Linux environment, a wide variety of publications are available, several of which are listed in the bibliography. The text, *Programming with GNU Software* (Loukides and Oram), is available from O'Reilly Publishing and is an excellent resource for the programmer who is just starting out with GNU tools (see bibliography references); also the Cygwin web site at www.cygwin.com has user documentation and the latest Cygwin information.

1.5.1 System Requirements (Windows Host Version)

1) Intel architecture (i586/i686) PC running one of the following Microsoft operating systems:
   - Windows XP Professional or Home Edition or newer
   - Windows 2000 Professional
   - Windows NT 4.0 and Windows 95/98 are no longer supported or recommended
2) CPU Clock Rate:
   - 500 MHz or higher
3) Minimum System RAM:
   - 256 MB
4) Free Disk Space:
   - 1 GB

1.5.2 System Requirements (Linux Host Version)

1) Intel architecture (i586/i686) PC running a Linux 2.x kernel.
2) CPU Clock Rate:
   - 500 MHz or higher
3) Minimum System RAM:
   - 256 MB
4) Free Disk Space:
   - 200 MB

See the appropriate target appendix for specific requirements by target. The i.MX GNU X-Tools™ toolsuite has been installed, built and tested on the following Linux distributions:

- Red Hat Linux Ver. 7.3
- Red Hat Linux Ver. 9.0
- Red Hat Enterprise Linux WS v3.3 or newer
- Linspire Ver. 3.5
- Debian Ver. 2.4.22

1.6 Compatible RTOS/OS Software

The i.MX GNU X-Tools has been used by customers and internal staff to build many of the most popular operating systems with board support packages, boot ROMs, network stacks and single/multithreaded applications since its first release in September 2000. A partial list follows:

- Accelerated Technology Nucleus RTOS and TCP/IP Stack
- EBSnet RTKernel RTOS, File System, and TCP/IP Stack
- Embedded Linux BSPs
- Express Logic ThreadX RTOS and TCP/IP Stack
- Interniche TCP/IP Stack
- Micrium u/COS-II RTOS and TCP/IP Stack
- MicroMonitor Boot ROM, by Ed Sutter
- Real-Time for Embedded Multiprocessor Systems (RTEMS) RTOS and TCP/IP Stack
- Quadros RTXC RTOS and TCP/IP Stack
- WindRiver vxWorks RTOS and TCP/IP Stack

1.7 Compatible JTAG/BDM Hardware

The i.MX GNU X-Tools has been used with a variety of JTAG/BDM debuggers. Although the following list is not comprehensive, we have high confidence in their ability to seamlessly integrate with the i.MX GNU X-Tools.

- Abatron BDI2000 with bdiGDB
- American Arium SC/LC JTAG Debuggers and SourcePoint
- EPI Tools MAJIC series
- Macraigor mpDemon/Raven/Wiggler
- Nohau EMUL-ARM
- Signum Systems Chameleon JTAG Debugger
- ARM Multi-ICE

Section 2. Install and Setup Procedures

There are two host versions of i.MX GNU X-Tools: Windows and Linux. Read the system requirements in Section 1.0 before proceeding further. The information in this Section gives the user details of installation, environment configuration, removal of toolsuite, setup testing, and troubleshooting:

- Section 2.1 Install i.MX GNU X-Tools on Windows Host
- Section 2.2 Uninstall i.MX GNU X-Tools on Windows host
- Section 2.3 Install i.MX GNU X-Tools on Linux Host
- Section 2.4 Uninstall i.MX GNU X-Tools on Linux Host
- Section 2.5 Setup test
- Section 2.6 Troubleshooting Guide for Installation Problems

2.1 Install i.MX GNU X-Tools on Windows Host

The following steps describe how to install the i.MX toolsuite. The README file on disk describes the contents of the i.MX GNU X-Tools distribution CD. Insure that the system requirements have been met, especially RAM and free disk space. On Windows NT/2000/XP hosts, login as administrator or have full permissions to the root directory of the destination disk volume.

Step 1.

Insert the Microcross i.MX GNU X-Tools distribution CD into the PC CD-ROM drive. On most systems the installer will auto-start the i.MX GNU X-Tools installation manager (setup.exe); if setup does not run, manually start the setup program by using Microsoft Explorer and double click on 'setup.exe', which is in the root CD directory.

Step 2.

After starting the setup program, follow the onscreen instructions for installing the program. We highly advise installing the Cygwin into a root volume of any local hard drive. The default setting is c:\Cygwin. Note: Two shortcuts will be installed on the Desktop; one for i.MX GNU X-Tools Shell and one for XWindows. XWindows is used for running the DDD debugger only, which is described in the section on How to Use GDB. Microcross' Visual GDB can be started from either the i.MX GNU X-Tools Shell or XWindows Shell.

Step 3.

Next, click on 'Exit' to exit the setup program. Go to Section 2.5 to perform the setup tests to verify a proper installation of i.MX GNU X-Tools. If for any reason the installation failed, go to Section 2.6 for troubleshooting tips.

Note: Once the toolsuite and Cygwin are installed, the toolsuite can be uninstalled and reinstalled without reinstalling Cygwin. This procedure may become necessary in the event that a toolsuite becomes corrupted or parts are accidentally deleted. Simply use the i.MX GNU X-Tools Shell and type 'xtools remove <target-alias>" (enter). To reinstall just the toolsuite, insert the i.MX GNU X-Tools CD into the CD-ROM , and from the i.MX GNU X-Tools Shell, type 'xtools install <target-alias>' (enter). Example:

```
$ xtools remove arm-elf  (enter)
$ xtools install arm-elf  (enter)
(Use the same CD-ROM drive that the original installation was made from.)
```

2.2 Uninstall i.MX GNU X-Tools on Windows Host

Uninstall i.MX GNU X-Tools and Cygwin

1) Start the Control Panel from the 'Start' menu.
2) Click on 'Add or Remove Programs'.
3) Scroll down to find 'i.MX GNU X-Tools with …' and click on 'Remove'.

The above remove procedure should remove the program and environment settings; however, in the event that anything should happen out of the ordinary, follow the next three steps to manually return your computer back to its pristine state.

To verify i.MX GNU X-Tools environment is uninstalled completely:

1) Run 'regedit' from the 'Start|Run' command
2) Look for the following Registry Keys:
   HKEY_LOCAL_MACHINE\Software\Cygnus Solutions
   HKEY_CURRENT_USER\Software\Cygnus Solutions

3) If they exist, manually delete them by selecting the key and pressing the Delete key.

To verify removal of environment variables:

1) Start the Control Panel from the 'Start' menu.
2) Click on 'System' and select the 'Advanced' Tab.
3) Click on 'Environment Variables'.
4) On 'System Variables' window, Click on 'Path' and then on the 'Edit' button.
5) Make sure all instances of 'Cygwin\bin', 'Cygwin\usr\bin', and 'Cygwin\usr\local\bin' along with the drive letter are removed.

Delete the Cygwin directory

1) Open Windows Explorer.
2) Click on the Cygwin directory and press the delete key.

2.3 Install i.MX GNU X-Tools on Linux Host

The following procedures is for installation of the i.MX GNU X-Tools toolsuite on a Linux host. Insure that the system requirements have been met, especially RAM and free disk space. The Linux installation procedure for i.MX GNU X-Tools (arm-elf toolsuite) is a command line procedure and is very simple and straightforward. We assume that you have a Linux OS running on your workstation or server before getting started with this installation. All commands are done from the Bash Shell (xterm). If you need the arm-linux (for GX-Linux builds) toolsuite installed, consult the GX-Linux User Guide for this toolsuite installation produre.

Step1.

Insert the i.MX GNU X-Tools distribution CD into the CD-ROM drive.

Step 2.

Start a Bash Shell (xterm) and login as 'root' (or use 'su'). You will need to do this each time you install or uninstall i.MX GNU X-Tools because you will need write access to the '/usr/bin' and '/usr/lib' directories where the toolsuite is installed.

Step 3.

Mount the i.MX GNU X-Tools distribution CD on the CD-ROM drive using the 'mount' command,

Example:
mount -t iso9660 /dev/cdrom /mnt/cdrom (enter)

The system will announce that it has mounted the CD-ROM as read-only. **Note:** On some Linux systems, the CD-ROM mounts by an automounter, which means step 3 may not be required.

Step 4.

Install the i.MX GNU X-Tools command tool from the Bash shell by executing the following commands:

cd / (enter)
tar –xvzf /mnt/cdrom/bin/rcfiles.tgz (enter)

At this point you should be able to execute the i.MX GNU X-Tools command tool inside of the Bash shell by its name. The command, 'xtools' (enter), should invoke the abbreviated tool help screen that shows the basic operating commands (see Section 4 for a complete description of the command tool).

Step 5.

To install an individual toolsuite, simply execute the command:

xtools install arm-elf (enter)

The command tool will extract the toolsuite tarball, installing all files in their appropriate locations. The install command assumes the default archive source path is '/mnt/cdrom/bin' using '/mnt/cdrom' as the CD-ROM mount point. If your default mount point is different, you can change the default value by editing '/usr/bin/xtools' file. If you would like to override the default path to the distribution medium, you may enter the command as 'xtools install arm-elf <arc-path>'. The 'arc-path' should always end in 'bin' as the tar archives are in the bin directory on the CD-ROM (e.g., 'xtools install arm-elf /mnt/cdrom1/bin').

Step 6.

If you do not have the 'tcl8.4/tk8.4' library installed in your Linux distribution (check your system documentation for details on how to make the check), here is the command to execute and install the library from Microcross' CD-ROM:

xtools install share-gdbtcl (enter) (add the 'arc-path' if not using the standard path described above)
Logout as root or 'su' at this time.

Step 7.

Install the test files by using the following commands (assuming the standard 'arc-path'):

```
$ cd /home  (enter)  (or other desired directory)
$ tar –xvzf /mnt/cdrom/src/test.tgz  (enter)
$ tar –xvzf /mnt/cdrom/src/bench++.tgz   (enter)  (If provided with the Target Toolsuite)
```

Go to Section 2.5 to perform a setup test of the toolsuite.

2.4 Uninstall i.MX GNU X-Tools on Linux Host

Uninstall i.MX GNU X-Tools Toolsuite

1) Start i.MX GNU X-Tools Bash Shell (xterm) and login as root or 'su'.
2) Type 'xtools remove arm-elf' to remove the toolsuite.

2.5 Setup Test

To perform a quick checkout and health status of the i.MX GNU X-Tools toolsuite installation, follow these steps to run the cross-compiler on a test directory under 'Cygwin/home/test' or '/home/test' on Linux (assuming you installed the 'test.tgz' tar file in this directory). For Windows users, start the i.MX GNU X-Tools Shell by double clicking on the desktop icon (red X on it), and issue the following commands in the shell. For Linux users, start an 'xterm' or Bash shell and issue these same commands:

```
$ xtools arm-elf  (enter)
arm-elf$ cd /home/test  (enter)  (or to the directory where the test files are located on the Linux host)
arm-elf$ ./run-all   (enter)   (dot forward slash run-all)
```

The cross-compiler will compile and run several test programs in the target Instruction Set Simulator (ISS), which is shown running in the shell. If there are errors of any kind, then there is a problem with the installation. Read the troubleshooting guide in Section 2.6 for additional information concerning possible problems and their solution.

If the run-all test passes, then go onto testing the C++ cross-compiler environment using Bench++. This next procedure could take up to **30 minutes** depending on the speed of your computer and the target toolsuite used. Bench++ is an AT&T C++ test suite that is shipped with the i.MX GNU X-Tools as a bonus test environment for toolsuites supported by Microcross. Bench++ tests most of the standard C++ language constructs in the latest GCC release and is a good indicator of a healthy i.MX GNU X-Tools C++ development environment. If you have any comments or suggestions to improve this test suite, please send your comments to support@microcross.com.

```
arm-elf$ cd /home/bench++  (enter) (or to the directory where the Bench++ test files are located on Linux
host)
arm-elf$ make clean all  (enter)
```

2.6 Troubleshooting Guide for Installation Problems

2.6.1 Windows / Cygwin Related Problems

Many developers have installed one version or another of Cygwin in the past and have experienced problems installing a newer version over the older version. The problems are most often due to an unclean uninstall. A user should cleanly uninstall other versions of Cygwin before installing the Microcross version. To cleanly uninstall Cygwin manually of all registry entries and path settings, follow these steps:

- Run ‘regedit’ from the ‘Start | Run’ command
- Look for the following Registry Keys:
    - HKEY_CURRENT_USER\Software\Cygnus Solutions
    - HKEY_LOCAL_MACHINE\Software\Cygnus Solutions

- If they exist, manually delete them by selecting the key and pressing the Delete key.

To Verify and / or Clean the Environment Variables, perform the following steps:

- Start Control Panel from the ‘Start’ menu.
- Click on ‘System’ and select the ‘Advanced’ Tab.
- Click on ‘Environment Variables’.
- On ‘System Variables’ window, Click on ‘Path’ and then on the ‘Edit’ button.
- Make sure all instances of ‘Cygwin\bin’, ‘Cygwin\usr\local\bin’ and ‘Cygwin\usr\bin’ along with the drive letter are removed.

2.6.2 Make.exe Related Problems

If the user has any version of Borland development tools on his/her machine, the user might experience a problem using the GNU Make. The reason there is a conflict is because Borland Make and GNU Make have the same tool name (make.exe). Microsoft’s make utility is called ‘nmake’; therefore, the user will not experience any problems with the Windows development tools from Microsoft. Another conflict has been noted with MKS application software. Both Borland and MKS conflicts can be easily fixed.

The user can fix the problem in one of two ways: 1) Edit the path settings in the ‘Advanced’ tab of the Windows ‘System Properties’ (in the ‘Control Panel’) and make sure that the Cygwin paths are before the Borland or MKS paths – move Cygwin paths to the left of the other program paths; or, 2) The user can remove the bothersome programs from the host machine. Obviously, the easiest and preferred method is to place the Cygwin paths to the left of the conflicting paths. Remember to separate each path and terminate with a ‘;’ semicolon.

2.6.3 Remove Carriage Returns in Source and Make Files

The user may experience compile errors if DOS style files are compiled directly with i.MX GNU X-Tools. This is due to the i.MX GNU X-Tools being configured for UNIX style file line feed termination instead of the DOS carriage return and line feed. Source files that have macro definitions and line continuation characters are always going to cause a compile error. If your source files are already in the DOS style format with carriage returns and line feeds, follow one of these steps to fix the source files; 1) If using Visual X-Tools, open the source files and execute a ‘Save As…’ from the File menu and select the ‘Line Format’ from the click-down text box at the bottom of the ‘Save As …’ dialog box – click on ‘Unix’ – do this for each source file and click ‘Save’; or 2) Open a BASH/GNU X-Tools Shell and navigate the command line (using Unix/Linux style commands) to the directory of the file you wish to clean the CRs out. Enter the following commands verbatim:

```
$ tr  -d   "\r"  <filename  >tmpfile  (enter)
$ rm  -f   filename  (enter)
$ mv  tmpfile  filename  (enter)
```

Note: the '<' and '>' are nececssary as part of the syntax. The 'filename' is the file you want to remove the contents, except for the carriage returns, and place the contents in a 'tmpfile'; then the next two procedures simply remove the original file and rename the 'tmpfile' to the orignal filename.

These three commands are necessary to remove the carriage returns from a DOS style source file, so that a user may use Cygwin to compile the source file into object code.

Generally, here are a couple of suggestions when using Cygwin and i.MX GNU X-Tools with source files. Besides carriage return issues with DOS files, do not put spaces in filenames and folder names – the user will experience problems with Unix style files in Cygwin as well as Linux/Unix. Use EMACS as your text editor on Windows using the xWin desktop icon to launch emacs; simply type ‘emacs’ after xWin starts a shell. EMACS is a Unix style editor that will not put carriage returns in your text files.

2.6.4 ‘xtools install’ Command Fails on Linux Host

Description

In some Linux versions the 'xtools install' command described in the i.MX GNU X-Tools install procedures fails claiming it cannot find unzip. Even though this problem has been fixed in the current ‘xtools’ procedure, we offer a remedy in case a future problem erupts. A Linux vendor may come out with a new product tomorrow and change the OSTYPE to an unknown name that ‘xtools’ does not recognize, so we offer the following procedure to insure users can resolve the problem themselves.

Resolution

The ‘xtools’ procedure utilizes the shell variable, called OSTYPE, to determine whether the current environment is Cygwin or Linux. We have discovered that different Linux distributions and versions set this variable differently, some using 'linux' and some using 'linux-gnu', for example. The xtools control script, '/usr/bin/xtools', checks the value of OSTYPE to determine which install procedure to use.

Procedure to Fix OSTYPE

- Follow the normal installation procedure as described in Sec 2.3. up through step 4.
- From a shell, change directory to '/usr/bin'. Enter the command 'echo $OSTYPE' and record the value of OSTYPE reported.
- Edit the file 'xtools' in '/usr/bin'. Go to the line that contains 'if[$OSTYPE=linux-gnu]; then' and edit it. If the reported OSTYPE is not 'linux-gnu', then the enter the appropriate reported OSTYPE. Save and exit the edit session. You should now be able to continue the installation process following the rest of the instructions.

Section 3. Quick Start Guide

3.1 i.MX LiteKit Setup

Windows Host Software Included in Home Installation

The Windows host solution package on Cywin contains all of the tools, utilities, and sources to perform testing, validation, experimentation, and application development for OS-less (single-thread) and embedded Linux environments. Here is a summary of the directories under 'Cygwin\home' that offer a quick start to understanding and working with the i.MX LiteKit.

Directory contents: …\Cygwin\home\

bench++	– C++ test benchmarks from AT&T for validating C++ tools
debug-example	– Macraigor usbDemon debug test example
gxlinux	– GX-Linux standard platform source directory
ocd-commander	– Macraigor OCD Commander installation software
splash	– splash binary and autostart script
test	– C example test code directory from Microcross
ucon	– uCon Serial Terminal Software (alternative to HyperTerminal)
umon	– MicroMonitor source directory
umon-recovery	– Recovery files for MicroMonitor
uwindows	– MicroWindows Minesweeper example
vxtools-example	– Visual X-Tools build and debug example

The i.MX LiteKit setup procedure follows and offers insight into the complete solution and how to get started using i.MX technology.

Requirements

- Host computer with Windows and i.MX GNU X-Tools pre-installed (see Section 2).
- Host computer must have either a DB-9 connector or USB-to-DB-9 adapter.
- HyperTerminal serial communications software or equivalent software (uCon is in the Cygwin/home/ucon directory if you desire to install it from the self-extracting exe file – we highly encourage using uCon over HyperTerminal – it is much easier to use and has special MicroMonitor hook features).
- Ethernet cross-over cable or hub connection to a LAN.
- 115 VAC Power.

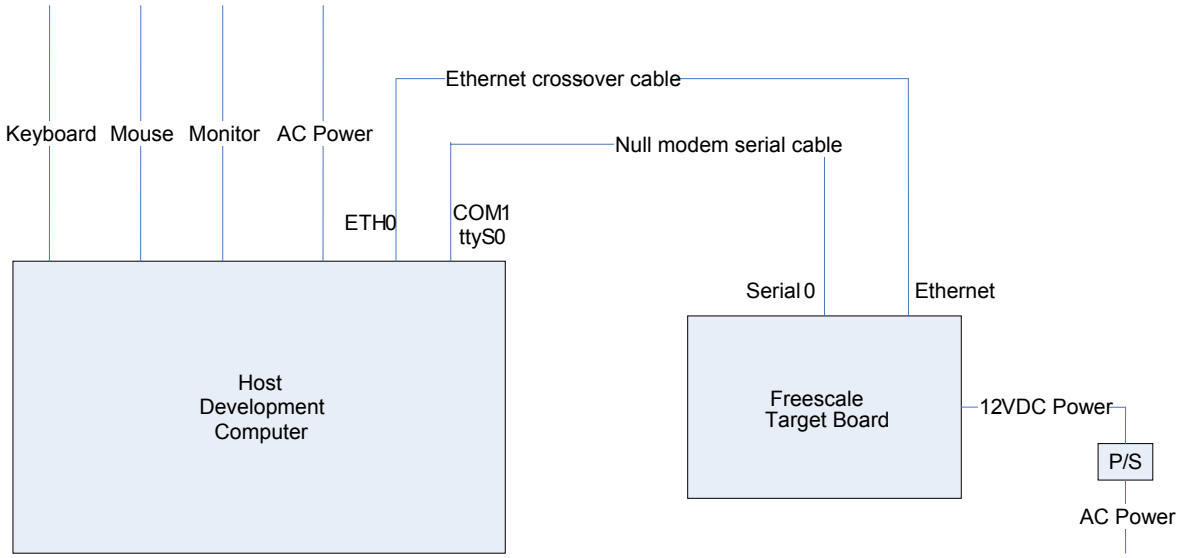
Step 1. Setup Serial Terminal

- Insure that the board is not powered on and your host computer is booted up.
- Connect the provided null-modem serial cable to the target board and host computer's DB-9 Serial Port or alternatively, .
- Configure your preferred Serial Terminal software with the following settings:

  38400 baud
  8 data bits
  No parity
  1 stop bit
  no flow control

- Connect power supply to the board, and the serial terminal should show the MicroMonitor startup banner and the 'uMON>' command prompt. Figure 3.1 shows a diagram of board connections to the host computer and power.

Figure 3.1 Diagram of Board Connections to Host Computer



Step 2. Setup uMON

You have two options to setup the network file, 'monrc' in MicroMonitor: manual settings or DHCP/BootP. Consult your system administrator if you need help in determining your network settings.

Manual network settings: The settings shown are *notional* and are for reference of the procedure only. Consult your network administrator if you do not understand how to setup network settings for your network. Use the uMON command line interface in your serial terminal session to enter these commands with your substituted settings.

```
uMON> set –c  (enter)
uMON> set IPADD 192.168.0.110   (enter)  (IP address)
uMON> set NETMASK 255.255.255.0  (enter)  (network mask)
uMON> set GIPADD 192.168.0.250  (enter)  (gateway IP address)
uMON> set –f monrc  (enter)
uMON> reset  (enter)
```

DHCP/BOOTP network settings: Follow this example and enter the commands using your serial terminal uMON command line interface.

```
uMON> set –c  (enter)
uMON> set IPADD DHCP (enter)  (or DHCPv or DHCPV or BOOTP)
uMON> set –f monrc  (enter)
uMON> reset  (enter)
```

Select only one for IPADD setting
DHCP No verbosity enabled
DHCPv Limited verbosity enabled
DHCPV Extreme verbosity enabled
BOOTP Runs BootP

IMPORTANT: After configuring the network settings and initiating the reset command, the target board will reboot MicroMonitor. Notice that in the serial terminal interface that the new IP address is displayed; if the new IP address does not show up on the display, go back to the previous step and make the necessary corrections to 'monrc'. When using DHCP/BootP, the new IP address should be annotated in your notes, because you will need this IP address for loading GX-Linux files via TFTP. If your DHCP/BootP server assigns a different IP address every time you reboot MicroMonitor, you may want to consider manually setting the network IP address while in the development phase of your project, because the IP address is a key parameter to your TFTP communications between the host development platform and the target board.

3.2 Build, Load, and Run Example Applications on MicroMonitor

Assumptions: All necessary steps in Section 3.1 are completed and the board is ready to communicate via TFTP. In addition, the LiteKit expansion/module board is connected to serial, Ethernet and powered on.

This section explains how to build, load and run a basic, single-threaded example application and MicroWindows' Minesweeper program on MicroMonitor. For the first example, the application code is located in 'Cygwin/home/umon/umon_apps/user_manual/main1.c' and provides a target-independent example of hooking up a 'Hello embedded world!' application to a target running MicroMonitor.

System dependent information is normally set in the makefile using a text editor to make changes; the makefile is located in 'Cygwin/home/umon/umon_apps/user_manual' directory'; however, note that we have already configured most settings for the LiteKit module with the exception of an IP address for your specific network settings. The following are all of the environment variables that normally need setting in the makefile. You can double check your board settings as a drill in this example or go onto the Example 1 below:

ARCH:
Set ARCH to one of the accepted CPU architectures (i.e. MIPS PPC, ARM, COLDFIRE).

MONCOMPTR:
Retrieve MONCOMPTR from the output of 'set'.

APPRAMBASE:
Set APPRAMBASE to the content of the APPRAMBASE shell variable or a bit higher.

TARGET_IP:
You can set TARGET_IP to the IP address of your target in the Makefile or on the command line as shown in Step 2 below.

Example 1, "Hello Embedded World" Build / Load / Run Procedure

Step 1.
Start a i.MX GNU X-Tools Shell using the red X icon on the desktop (assuming the toolsuite is installed).

Step 2.

Issue these commands in the Shell:

$ cd /home/umon/umon_apps/user_manual (enter)
$ make app1 (enter) (builds application)
$ make TARGET_IP=<board-IP> app1dld (enter) (downloads application to the board)
Substitute your board IP address above.
Example: $ make TARGET_IP=192.168.0.110 app1dld (enter) (no spaces in 'app1dld')

Step 3.

In the Serial Terminal at the 'uMON>' command line prompt, issue the following command to run the application:

uMON> app1 (enter) (run the example application)
uMON> Hello embedded world! (prints out from the example program)
uMON> tfs ls (enter) (shows filesystem contents)
uMON> tfs rm app1 (enter) (removes/deletes app1 from the filesystem)

Each of the five example applications can be built, loaded and run in the manner described above (i.e., app1, app2, app3, app4, and app5). Some example applications take arguments that are passed in at execution (e.g., app2 arg1 arg2 arg3). The MicroMonitor User Manual is a great resource to learn more – see Cygwin\docs directory.

Example 2, Minesweeper Build / Load / Run Procedure

The following steps describe the procedure to build a MicroWindows application called Minsweeper. This example is a complete keyboard and visual control of the i.MX LiteKit board video system.

Step 1.

Start a i.MX GNU X-Tools Shell using the red X icon on the desktop. Connect your target board to a network or a cross-over Ethernet cable and power it on. Ensure you have a target IP address assigned to the board. On your host machine, issue these commands in the i.MX GNU X-Tools Shell:

$ cd /home/uwindows/src/demos/elsdemo (enter)
$ make TARGET=<i.MX-board> TARGET_IP=<board-IP> mine (enter)
(builds application and downloads it into the target board)
TARGET options are 'CSB535FS' for i.MX21 and 'CSB536FS' for i.MXL
Example:
$ make TARGET=CSB535FS TARGET_IP=192.168.0.110 mine (enter)
The program builds and downloads to the target board automatically (assuming you have the target board connected and powered on).

Step 2.

At the Serial Terminal 'uMON>' command line prompt, issue the following command to run the application

uMON> mine (enter)

Follow the onscreen instructions to play minesweeper. Cycle power to quit the game and reboot to MicroMonitor.

3.3 Build, Load, and Run GX-Linux and Example Applications

Assumptions: All necessary steps in Section 3.1 are completed and the board is ready to communicate via TFTP. In addition, the LiteKit expansion/module board is connected to serial, Ethernet and powered on.

This section explains how to build and load applications into the target Freescale development board using the Windows/Cygwin host provided by Microcross on the Standard GX-Linux Platform. Since kernel and library

building are not provided on a Windows/Cygwin host, Microcross pre-built the kernel and libraries on a Linux host and installed them on the Windows/Cygwin host for those who want to evaluate GX-Linux, build applications and run them.

Step 1. Make All

Start a i.MX GNU X-Tools Shell by clicking on the desktop icon installed. If there is no icon, you may not have the Application Development Platform on Windows installed correctly – go back to Section 2 and follow the installation instructions.

Make the example programs and create a ROM image for the 'romfs.img' filesystem by issuing the following commands:

```
$ cd /home/gxlinux (enter)
$ make all  (enter)
(builds example applications, installs them, and re-generates a ROM filesystem)
```

Step 2. Load Binaries and Startup Script

At this point, we assume that the target board is powered up, connected to Ethernet, and the uMON prompt is displayed in a serial terminal. If not in this configuration, go back and get the board ready. From the i.MX GNU X-Tools Shell and assuming you are still in the *build* directory, issue the following command that will load the GX-Linux binaries and startup script.

```
$ make TARGET_IP=<board-IP>  load  (enter)
Example
$ make TARGET_IP=192.168.0.110  load  (enter)
```

Three images should start uploading to the board – see the serial terminal interface for upload response in MicroMonitor. The images loaded are as follows: 1) 'zImage' (kernel); 2) 'romfs.img' (ROM root filesystem); and 3) 'startlinux' (startup script).

Step 3. Boot Linux

From the serial terminal window, issue the following command:

```
uMON> startlinux (enter)  (this script calls the Linux kernel and filesystem to execute)
```

The Linux kernel should uncompress and mount a ROM filesystem. If there are any errors, recycle power on the board and repeat this step. Occasionally the kernel aborts on a bootup due to a corrupted flash file system or missing romfs.img file. Follow steps in the MicroMonitor tips section to issue a command (uMON> tfs clean) to defrag the flash file system manually.

The Linux startup script is set with a Tiny Filesystem attribute that makes 'startlinux' an executable like a batch file in DOS. To change this attribute so that you can automatically boot into Linux with a query to stop after MicroMonitor boots up, issue the following command after boot up of MicroMonitor:

```
uMON> tfs –feB cp startlinux startlinux  (enter)
```

WARNING: '-feB' is case sensitive and if you use a lower case 'b' in the switch option, you will not be able to boot into uMON again without re-flashing through the JTAG, which is complicated and time consuming. To change the

attribute back to manual boot, substitute '-fe' in place of '-feB' in the above command above. To stop the automatic boot, press any key while uMON is booting and before Linux boots. The bootup will stop at the uMON command prompt.

Step 4. Run Example Applications in GX-Linux

When the GX-Linux boots to a Linux command shell prompt, you can enter the following commands in the serial terminal to get acquainted with all of the example programs loaded in the '/usr/bin'; each example file has a '.x' file extension. In the serial terminal command line interface window, issue the following commands:

```
# cd /usr/bin     (enter)
# ls              (enter)  (shows all examples; to execute examples, perform the next step)
# hello.x         (enter)  (executes hello world program)
```

There are approximately 20 other programs that can be executed in the same manner. Tryout as many of the examples as you wish. Read the next section to become more familiarized with MicroMonitor.

Step 5. How to Add Your Own Applications

The fastest way to get your own Linux application on the target board is to copy your application – named with a '.x' file extension -- into the examples directory, which is under the build directory, and issue the following command in the i.MX GNU X-Tools:

```
$ make install-apps romfs (enter)
$ make load               (enter)  (reload your images to the target board)
```

After uploading three files to the target board (startlinux, zImage, and romfs.img), you can manually start GX-Linux by issuing the following commands in the serial terminal (uCon / Hyperterminal):

```
uMON> (enter)              (get a command prompt after upload)
uMON> reset (enter)        (get the board in a refresh state)
uMON> startlinux (enter)   (Linux boots to a user prompt '#')
```

On the target board in the '/usr/bin' directory are example programs, including the one you just created and copied into the examples directory. Issue the following commands:

```
#  cd /usr/bin (enter)
#  ls (enter)
```

You can see all examples; they have a '.x' file extension on them. To run any example, issue the by-name command:

```
#  double.x (enter)
#  float.x  (enter)
#  hello.x  (enter)
etc…
```

Step 6. Useful Linux Commands

```
# cat /proc/cpuinfo
Show information such as bogomips, architecture, and manufacture information.

# reboot
```

Reboots system into uMON.

Microcross has a complete Professional Platform edition of GX-Linux that has all of the board drivers, documentation, and total rebuild environment with master makefile and components of a complete board support package. For those serious users of Embedded Linux, the GX-Linux Professional Platform edition is a necessary and affordable time saver to jump start your project with Linux. As required by the General Public License (GPL), all GNU sources are available on the distribution CD-ROM in the 'src' directory. Contact Microcross for details on how you can get a discount by already being a Freescale customer.

3.4 Debug with usbDemon Example

First-Time User of usbDemon

IMPORTANT: The first time user will need to install OCD Commander to get the right USB drivers installed on the host machine. Skip to 3.4.1 if OCD Commander and USB drivers were previously installed. To install OCD Commander, open an Explorer file browser on your host machine and navigate to 'Cygwin\home\ocd-commander' directory – i.MX GNU X-Tools Windows host version must be already installed. Make sure that the usbDemon USB cable is disconnected before installing OCD Commander. Execute the 'ocd-commander.exe' installation utility by double clicking on the filename and follow the on-screen instructions for installing OCD Commander and USB drivers. After installation, reboot the host machine so that the new drivers are available to install. Connect the LiteKit provided USB cable between the host computer and target module. The Windows operating system should detect the USB connection and prompt the user to install the appropriate drivers.

The host computer should respond with 'Found New Hardware Wizard' dialog box. Select 'Install the Software Automatically' and click on Next. Click on 'Continue Anyway' on the 'Hardware Installation' dialog. Click on 'Finish' to complete the first of two drivers installation procedure.

The second 'Found New Hardware Wizard' will pop up, and you will need to repeat exactly the same inputs as described above for installing the first driver. Now that both drivers are installed, the usbDemon is now useable from this host machine.

Double click on the desktop icon for 'usbDemon Finder', which was installed with OCD Commander. The usbDemon finder will display the device number assigned to the usbDemon that is connected. Write this device number down for future reference when we activate the OCDRemote. Close the 'usbDemon Finder' and proceed to the example program.

3.4.1 Example Program

Under the Cygwin directory, '/home/debug-example', there is an example LiteKit program that can be debugged in a remote target session using GDB/Visual GDB and the built-in Macraigor usbDemon JTAG interface. Note: The Macraigor USB devices are currently not supported under Linux.

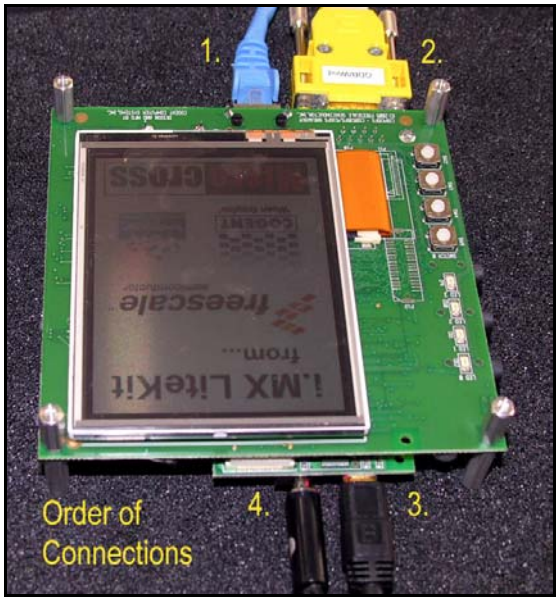
The following files are included in '/home/debug-example' directory under Cygwin:

.gdbinit	GDB Initialization File
crt0.S	Assembly source for C initialization code
ldscript	Linker script
Makefile	Compiles and links source files using i.MX GNU X-Tools
README	The readme file
test.c	C source for example program
testsub.c	C subroutine source for example program

Assumptions: OCD Commander and its utility, usbDemon Finder, are installed before starting this Demo Program. To find out what USB device number to use, execute the usbDemon Finder utility using the desktop icon and note the device number of the usbDemon (the USB cable has to be connected and the board while powered on in order to get this device number).

To Run the example on the LiteKit board you need to make connections to the host computer USB connection with the cable provided by Freescale and then to the board in the order shown in Figure 3.2. Next, follow the steps below.

Figure 3.2 Order of Connections



Step 1.

Construct an example application: On Windows host, start a i.MX GNU X-Tools Shell and then enter these commands:

```
$ cd /home/debug-example    (enter)
$ make    (enter)
```

Step 2.

Make connections to the target LiteKit board (USB cable, null-modem serial cable, Ethernet cable, and power cable); turn power on to the board, and start OCDRemote in a second i.MX GNU X-Tools Shell.

Always prior to running GDB/Visual GDB you must first start OcdRemote. This stand alone utility listens to a TCP/IP port and converts incoming GDB commands to JTAG signals. Here is the standard Macraigor command line format:

```
<GDB>--TCP/IP port--<OcdRemote>--LPTx-<MacraigorDevice>-JTAG--<ARM920T CPU>
                                                          --USBx-
                                                          --COMx-
```

OCDRemote takes additional OPTIONAL parameters that allows you to modify it's modes of operation:

```
OCDRemote -c <CPU type> [,<CPU type>],..] [-p <port number>]
          [-d <device>] [- a <device address>] [-s <speed>]

where: <CPU type> = ARM920T        ... for i.MXL
       <CPU type> = ARM926EJ-S  … for i.MX21

     (enter "ocdremote" for a complete CPU list if you desire)
   <port number> = tcp/ip port, default = 8888
   <device> = WIGGLER, RAVEN,MPDEMON_SERIAL, MPDEMON_PARALLEL,
                    MPDEMON_ETHERNET, USB, default = RAVEN
            <device address> =
                  WIGGLER/RAVEN/MPDEMON_PARALLLEL: LPT number 1 - 4
        MPDEMON_SERIAL : COM number : 1 - 4
        MPDEMON_ETHERNET : tcp/ip address in the format :
                                              xxx.xxx.xxx.xxx
        USB : macraigor USB device number : 0 - 15
             default = 1
               <speed> = JTAG clock rate 1 - 8, default = 1 (fastest)
                     (see GNU TOOLS FAQ page for a device speed vs
                                    jtag clock rate table)
```

Here is an example:

In the second i.MX GNU X-Tools Shell, enter the following command line (assuming the usbDemon is assigned device 0 at JTAG speed 1 [6 Mhz]):

```
$ ocdremote -c ARM920T -d USB -a 0 -s 1  (enter)
```

Feedback from the computer (assuming the USB cable is connected and the LiteKit board is powered on) is the following:

```
“JTAG SDO <-| CPU(1) ARM920T : listening on port 8888 |<- JTAG SDI”
```

Important: Each time you connect and disconnect GDB/Visual GDB, you must repeat this step. The Socket automatically shuts down when GDB/Visual GDB disconnects.

Step 3.

Start either GDB or Visual GDB (GDB with a GUI interface) to start download and debug.

Command Line GDB Example

In the first Shell, enter the following (assuming you are still in the same directory that contains the test image file built by make -- /home/debug-example):

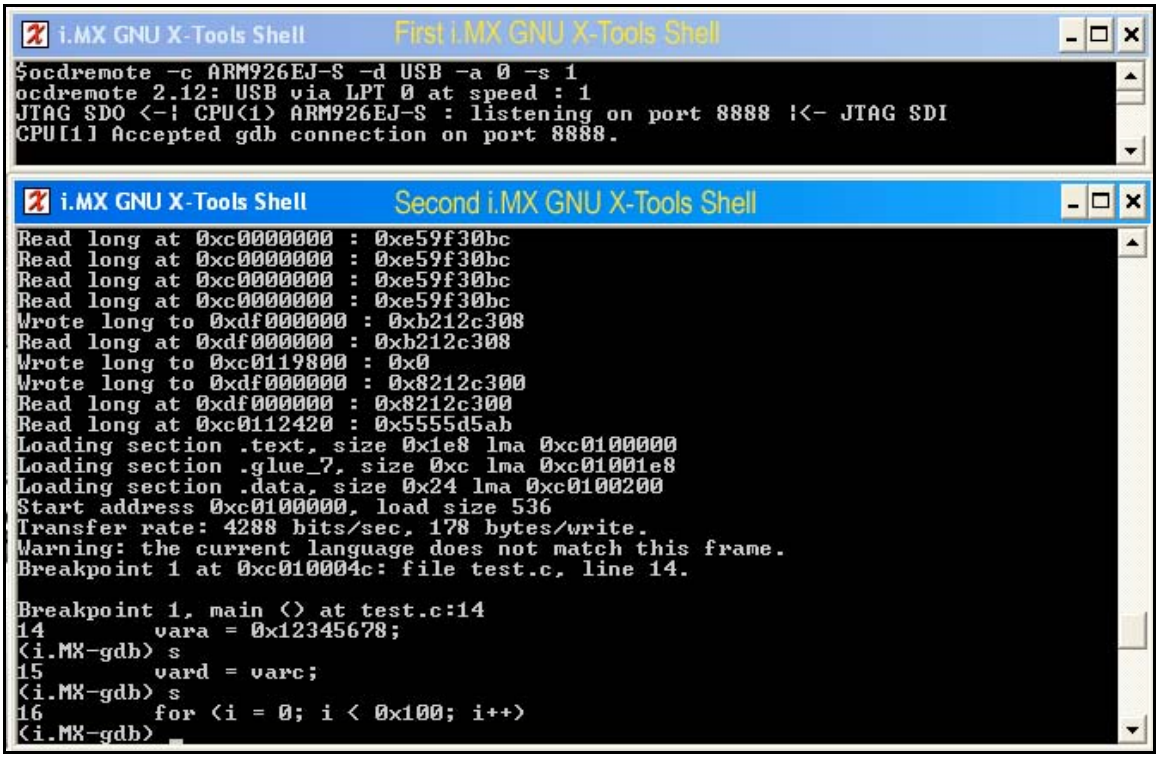
```
$ arm-elf-gdb test.x  (enter)
```

This command will run the command line version of GDB, load the program (using .gdbinit) and stop at a breakpoint on main(). To single step or perform other GDB commands, review the list of GDB commands below. Here are some examples:

(i.MX-gdb) i r	(enter) (shows register contents)
(i.MX-gdb) s	(enter) (single step)
(i.MX-gdb) c	(enter) (continue to next breakpoint or end of program)
(i.MX-gdb) ctrl-C	(enter) (exit debug)
(i.MX-gdb) quit	(enter) (exit GDB)

Figure 3.3 below shows two i.MX GNU X-Tools Shells. The first i.MX GNU X-Tools Shell shows the usbDemon setup of ocdremote. The second i.MX GNU X-Tools Shell shows the loading of the example program and breaking at main, and then two manual single steps through the program using 's' to perform the steps.

Figure 3.3 Command Line Debugging


i.MX GNU X-Tools Shell
First i.MX GNU X-Tools Shell
$ocdremote -c ARM926EJ-S -d USB -a 0 -s 1
ocdremote 2.12: USB via LPT 0 at speed : 1
JTAG $DO <-¦ CPU(i) ARM926EJ-S : listening on port 8888 ¦<- JTAG SDI
CPU[1] Accepted gdb connection on port 8888.
i.MX GNU X-Tools Shell
Second i.MX GNU X-Tools Shell
Read long at 0xc0000000 : 0xe59f30bc
Read long at 0xc0000000 : 0xe59f30bc
Read long at 0xc0000000 : 0xe59f30bc
Read long at 0xc0000000 : 0xe59f30bc
Wrote long to 0xdf000000 : 0xb212c308
Read long at 0xdf000000 : 0xb212c308
Wrote long to 0xc0119800 : 0x0
Wrote long to 0xdf000000 : 0x8212c300
Read long at 0xdf000000 : 0x8212c300
Read long at 0xc0112420 : 0x5555d5ab
Loading section .text, size 0x1e8 lma 0xc0100000
Loading section .glue_7, size 0xc lma 0xc01001e8
Loading section .data, size 0x24 lma 0xc0100200
Start address 0xc0100000, load size 536
Transfer rate: 4288 bits/sec, 178 bytes/write.
Warning: the current language does not match this frame.
Breakpoint 1 at 0xc010004c: file test.c, line 14.
Breakpoint 1, main () at test.c:14
14 vara = 0x12345678;
(i.MX-gdb) s
15 vard = varc;
(i.MX-gdb) s
16 for (i = 0; i < 0x100; i++)
(i.MX-gdb)

Visual GDB Example

Before getting started, reset ocdremote as you did in Step 2 in the previous example. You need ocdremote in the listening mode before connecting with Visual GDB.

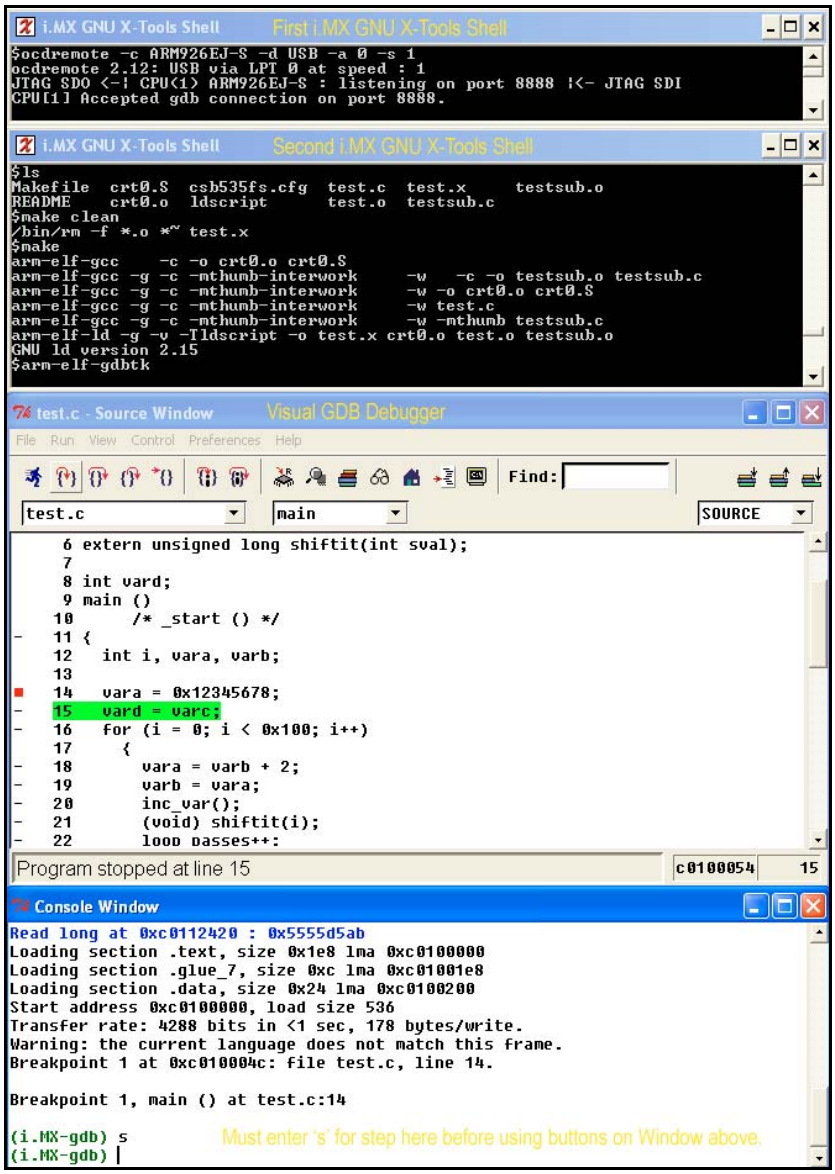
In the first Shell, enter the following command assuming you are still in the same directory that contains the test image file built by make -- /home/debug-example):

$ arm-elf-gdbtk test.x

This command will run Visual GDB, load 'test.x' and stop at breakpoint on main(). If the target settings dialog pops up, select Target='Remote/TCP', Hostname='localhost', and Port='8888' -- click on OK. Each subsequent run will automatically remember these settings until you change them again. To single step or perform other GDB commands, you may use the visual buttons that reflect equivalent functions as the command line version or you may enter the command line version in the console Window (click on 'View|Console' if it is not viewed upon start of Visual GDB). To exit, click on 'File|Exit' on the menu.

Figure 3.4 shows a Visual GDB on-chip debug session with Console Window visible. The i.MX GNU X-Tools shell shows ocdremote startup. The second i.MX GNU X-Tools shell shows the making of the example and startup of the Visual GDB debugger. No file was specified on the command line because .gdbinit file contains the load command. After starting Visual GDB, the Console Window and GUI start. The first command in the Console is 's' and is needed to get control over the GUI after setting the first breakpoint. From here you can single step all the way through the program, view registers, view memory, etc..

Figure 3.4 Visual GDB On-Chip Debug Session


i.MX GNU X-Tools Shell
First i.MX GNU X-Tools Shell
$ocdremote -c ARM926EJ-S -d USB -a 0 -s 1
ocdremote 2.12: USB via LPT 0 at speed : 1
JTAG SDO <-! CPU(1) ARM926EJ-S : listening on port 8888 !<- JTAG SDI
CPU[1] Accepted gdb connection on port 8888.
i.MX GNU X-Tools Shell
Second i.MX GNU X-Tools Shell
$ls
Makefile crt0.S csb535fs.cfg test.c test.x testsub.o
README crt0.o ldscript test.o testsub.c
$make clean
/bin/rm -f *.o *~ test.x
$make
arm-elf-gcc -c -o crt0.o crt0.S
arm-elf-gcc -g -c -mthumb-interwork -w -c -o testsub.o testsub.c
arm-elf-gcc -g -c -mthumb-interwork -w -o crt0.o crt0.S
arm-elf-gcc -g -c -mthumb-interwork -w test.c
arm-elf-gcc -g -c -mthumb-interwork -w -mthumb testsub.c
arm-elf-ld -g -v -Tldscript -o test.x crt0.o test.o testsub.o
GNU ld version 2.15
$arm-elf-gdbtk
test.c - Source Window
Visual GDB Debugger
File Run View Control Preferences Help
Find:
test.c
main
SOURCE
6 extern unsigned long shiftit(int sval);
7
8 int vard;
9 main ()
10 /* _start () */
11 {
12 int i, vara, varb;
13
14 vara = 0x12345678;
15 vard = varc;
16 for (i = 0; i < 0x100; i++)
17 {
18 vara = varb + 2;
19 varb = vara;
20 inc_var();
21 (void) shiftit(i);
22 loop_passes++;
Program stopped at line 15
c0100054
15
Console Window
Read long at 0xc0112420 : 0x5555d5ab
Loading section .text, size 0x1e8 lma 0xc0100000
Loading section .glue_7, size 0xc lma 0xc01001e8
Loading section .data, size 0x24 lma 0xc0100200
Start address 0xc0100000, load size 536
Transfer rate: 4288 bits in <1 sec, 178 bytes/write.
Warning: the current language does not match this frame.
Breakpoint 1 at 0xc010004c: file test.c, line 14.
Breakpoint 1, main () at test.c:14
(i.MX-gdb) s
Must enter 's' for step here before using buttons on Window above.
(i.MX-gdb)

Construction of the '.gdbinit' File

In the .gdbinit file are setup calls to the JTAG. The following two lines are commented out, but are available to users of large files that need debugging.

```
set remote memory-write-packet-size 1024
set remote memory-write-packet-size fixed
```

These increase the frame size that GDB uses to communicate with OcdRemote. These lines are not necessary but increase the download speed significantly. GDB will prompt you to approve this change in the packet size. enter yes.

If GDB is run in GUI mode the .gdbinit file leaves the GUI in the "unattached" state, i.e., the execution control buttons are grayed out. This state can be cleared by opening a console window and issuing a "step" command. The execution control buttons will then be made active.

The "RUN" command (man running icon) does NOT work with a remotely connected target. Use the "CONTINUE" (->{} icon) command to invoke the program to run (even from the begining).

3.4.2 OCDRemote Monitor Commands

Monitor commands implement various functions that are not available using GDB directly and vary from CPU type to CPU type. You can get the full list of 'monitor' commands by issuing the command 'monitor help' in the console window after GDB has attached to OcdRemote.

```
monitor help
monitor allrun
monitor allstop
monitor char/short/long <addr>
monitor char/short/long <addr> = <val>
monitor endian [<big|little>]
monitor halt
monitor reg <regname>
monitor reg <regname> = <value>
monitor reset
monitor resetrun
monitor runfrom <addr>
monitor set memspace <virtual|physical|#>
monitor set cpu <cpu number>
monitor set/clear hbreak [<address>]
monitor set regbufaddr <address>
monitor sleep <seconds>
monitor status
monitor sync cpus <on/off>
```

These commands are availble for all CPU types. They can be executed either from within a '.gdbinit' file or from from the GDB console window.

3.4.3 Some Useful GDB Console Commands

Here are some useful text mode commands to to use in the GDB console window. If you use Visual GUI then all of these commands are available using the pull downs or buttons.

s Step, single step a C source code line.

si Step Instruction, single step a machine code instruction.

c Continue, run the processor after a step or breakpoint.

b Breakpoint, set a breakpoint at specified location.

^C Control-C, stop execution from the keyboard. For Visual GDB (the GUI) this will only work under a Linux host; to stop the program under Cygwin, you need to press the 'Stop' button.

l List, show the source code being executed.

x Examine, show the contents of memory.

i r Info registers, show the contents of all registers.

set Set, change the contents of ram or a register.

3.5 Visual X-Tools Example

The following steps describe the operation of the Visual X-Tools with an example application program for the i.MX LiteKit. The example program was created with the Visual X-Tools wizard and is setup to work with Visual GDB and the Macraigor usbDemon that is built into the LiteKit module.

The following files are included in '/home/vxtools-example' directory under Cygwin:

.gdbinit	GDB Initialization File for the i.MX LiteKit board
crt0.S	Assembly source for C initialization code
ldscript	Linker script
test.c	Main C source file for example program
test.mak	Autogenerated/maintained GNU style makefile for Visual X-Tools
test.vpwhist	Project history file
test.vtg	Tag database file
test.vxp	Visual X-Tools Project file
test.vxw	Visual X-Tools Workspace file
testsub.c	C subroutine source for example program
Debug/crt).o	Object file for startup rountine
Debug/test.o	Object file for test.c source file
Debug/test.x	Executable ELF image with debug symbols
Debug/testsub.o	Object file for testsub.c source file

Assumptions: Visual X-Tools IDE is installed (free 30-day evaluation is available on the i.MX GNU X-Tools CD-ROM). To install Visual X-Tools, insert the i.MX GNU X-Tools CD-ROM, open Windows Explorer and navigate to the 'vxtools' directory. Double click on 'setup.exe' and follow the onscreen instructions to install Visual X-Tools. A user guide is also available in this directory. The other assumptions include, i.MX GNU X-Tools are installed on Windows host, the i.MX LiteKit board is connected, including serial, Ethernet, usbDemon, and power. Moreover, the OCD Commander and its utility, usbDemon Finder, are installed. To find out what USB device number to use, execute the usbDemon Finder utility using the desktop usbDemon Finder icon and note the device number of the usbDemon (the USB cable has to be connected to the board while powered on in order to get this device number).

Step 1.

Startup Visual X-Tools IDE from a desktop icon. See the Visual X-Tools IDE icon in Figure 3.5.

Figure 3.5 Visual X-Tools IDE Icon

Step 2.

Click on 'Project|Open Workspace…' from the menu. Navigate to the following directory:

../Cygwin/home/vxtools-example

Open the workspace file 'test.vxw' and go onto the next step.

Step 3.

On the Visual X-Tools menu click on 'Build|Set Active Configuration…' and select 'Debug'.

Step 4.

On the Mini Toolbar (see Figure 3.6), click on 'Clean & Build Project' icons.

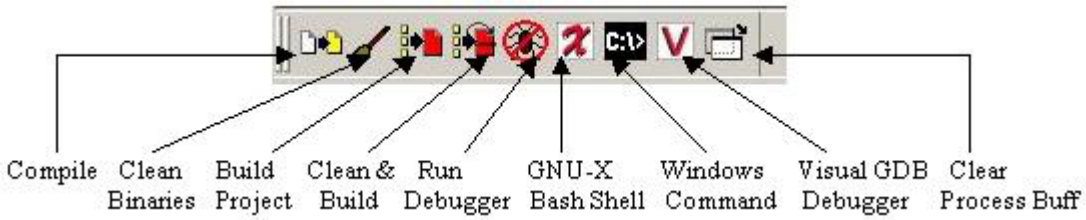


Figure 3.6 Mini-Toolbar Description

These commands are also located under the 'Build' menu. The Build window at the bottom of Visual X-Tools shows the verbose output of the build. After completion of the build, go to the next step.

Step 5.

Click on the i.MX GNU X-Tools Bash Shell icon (red X icon in Figure 3.6). Make connections to the target LiteKit board (USB cable, null-modem serial cable, Ethernet cable, and power cable); turn power on to the board, and start OCDRemote in a second i.MX GNU X-Tools Shell.

Always prior to running GDB/Visual GDB you must first start OcdRemote. This stand alone utility listens to a TCP/IP port and converts incoming GDB commands to JTAG signals. Here is the standard Macraigor command line format:

<GDB>--TCP/IP port--<OcdRemote>--LPTx-<MacraigorDevice>-JTAG--<ARM920T CPU>
--USBx-
--COMx-

OCDRemote takes additional OPTIONAL parameters that allows you to modify it's modes of operation:

```
OCDRemote -c <CPU type> [,<CPU type>],..] [-p <port number>]
          [-d <device>] [- a <device address>] [-s <speed>]

where: <CPU type> = ARM920T       ... for i.MXL
       <CPU type> = ARM926EJ-S  … for i.MX21
       <port number> = tcp/ip port, default = 8888
       <device> = WIGGLER, RAVEN,MPDEMON_SERIAL, MPDEMON_PARALLEL,
                  MPDEMON_ETHERNET, USB, default = RAVEN
            <device address> =
                 WIGGLER/RAVEN/MPDEMON_PARALLLEL: LPT number 1 - 4
          MPDEMON_SERIAL : COM number : 1 - 4
          MPDEMON_ETHERNET : tcp/ip address in the format :
                                              xxx.xxx.xxx.xxx
          USB : macraigor USB device number : 0 - 15
               default = 1
            <speed> = JTAG clock rate 1 - 8, default = 1 (fastest)
                 (see GNU TOOLS FAQ page for a device speed vs
                                  jtag clock rate table)
```

Here is an example:

In the GNU X-Tools Shell, enter the following command line (assuming the usbDemon is assigned device 0 at JTAG speed 1 [6 Mhz] and we are using the i.MXL target processor – we could have substituted ARM926EJ-S for the i.MX21 target processor):

```
$ ocdremote -c ARM920T -d USB -a 0 -s 1  (enter)
```

Feedback from the computer (assuming the USB cable is connected and the LiteKit board is powered on) is the following:

```
“JTAG SDO <-| CPU(1) ARM920T : listening on port 8888 |<- JTAG SDI”
```

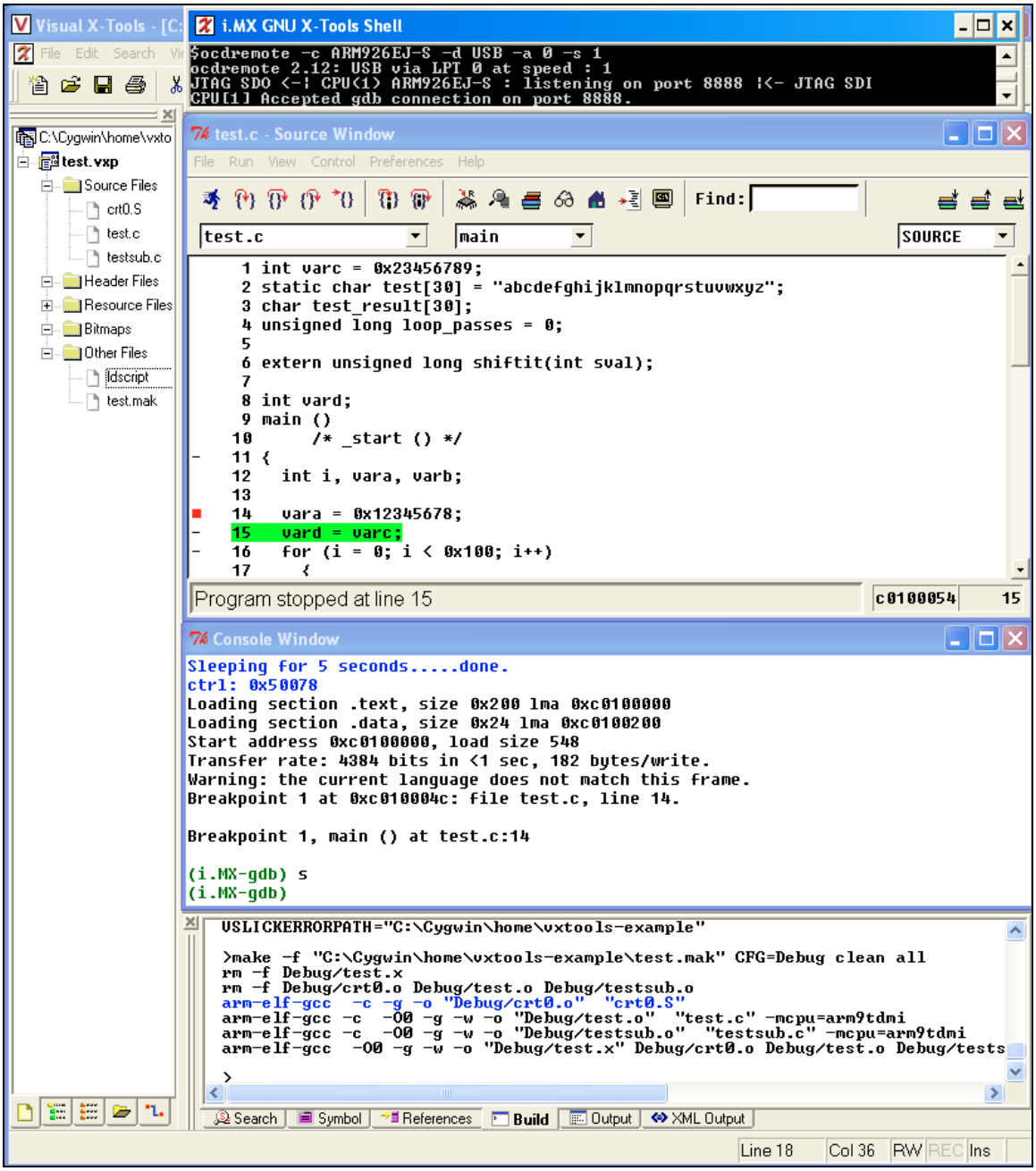
Important: Each time you connect and disconnect Visual GDB, you must repeat this step. The Socket automatically shuts down when GDB/Visual GDB disconnects.

Step 6.

Click on Visual GDB Debugger icon on the Mini-Toolbar (see Figure 3.6 above). This command will run Visual GDB, load ‘test.x’ and stop at breakpoint on main(). If the target settings dialog pops up, select Target='Remote/TCP', Hostname='localhost', and Port='8888' -- click on OK. Each subsequent run will automatically remember these settings until you change them again. To single step or perform other GDB commands, you may use the visual buttons that reflect equivalent functions as the command line version or you may enter the command line version in the console Window (click on ‘View|Console’ if it is not viewed upon start of Visual GDB). To exit, click on ‘File|Exit’ on the menu. Figure 3.4 above shows a Visual GDB on-chip debug session with Console Window visible. The Console Window needs to be started to get control over the GUI after setting the first. Enter ‘s’ for step in the Console window to enable all of the GUI functions on the Visual GDB. From here you can single step all the way through the program, view registers, view memory, etc.. To get more

specific information on how to use Visual X-Tools, consult the Visual X-Tools User Guide. Figure 3.7 shows Visual X-Tools, the i.MX GNU X-Tools Shell with ocdremote, and Visual GDB debugger all operating in the debug mode.

Figure 3.7 Visual X-Tools Debug Example



--- END OF EXAMPLE PROGRAM ---

3.6 LCD Tests

To test the LCD for color and touch screen use, there are two simple programs that are preconfigured for this purpose in MicroMonitor. Follow the simple procedures below to test the system with these programs.

Assumptions: The LiteKit board is connected to a host serial terminal and powered on.

Step 1.

In a serial terminal shell with MicroMonitor booted, issue the following command:

uMON> lcd_tst (enter)

Input any key to change the LCD color (repeat). To exit, press 'x' key.

Step 2.

In the same terminal shell with MicroMonitor booted to a prompt, issue the following command (note: this example is only working on the i.MXL LiteKit at this time):

uMON> ads (enter)

Press on the LCD screen at various points and observe the serial terminal display. Computations of pen up and pen down (X and Y parameters) will display on the screen. To exit, press 'x' key.

This concludes the simple LCD tests.

3.7 Rebuild MicroMonitor and Reflash Board

3.7.1 Normal Reflash Procedure

In the event that you want to modify MicroMonitor and reflash the board with the newly built MicroMonitor, here is the procedure to rebuild, load and run the new MicroMonitor image. This procedure assumes you have a working MicroMonitor in flash; If for any reason that MicroMonitor becomes corrupted in flash and will not boot correctly, you will need to use the JTAG port on the expansion board or the usbDemon on the module board to load a RAM image and use it to reflash memory with a binary image. Follow the procedure in Section 3.5.2 below. Otherwise, use the following procedure to reflash MicroMonitor.

Step 1.
Open a i.MX GNU X-Tools Shell.

Step 2.
Issue the following commands:

$ cd /home/umon/umon_ports/csb536fs (enter) (for MXL port – substitute 'csb535fs' for MX21)
$. bashrc (enter) (dot space 'bashrc' sets dependency paths)
$ make rebuild (enter) (cleans and rebuilds binary images)
$ make TARGET_IP=xxx.xxx.xxx.xxx newmon (enter) (substitute actual IP address)

Wait for approximately 30 seconds and power cycle the board to restart MicroMonitor. You will be prompted to input the last four digits of the MAC address. Use the last four digits on the label that is on the board and separate the every two digits with a ':' colon. Confirm the new MAC address when prompted, and you are done.

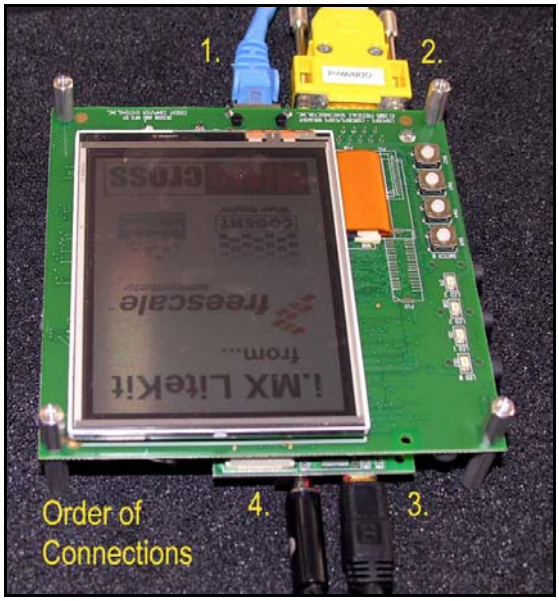
3.7.2 Dissaster Recovery Procedure

Perform this procedure **ONLY** if the module board on-flash has become corrupted and MicroMonitor fails to bootup. Multiple use (beyond 1-2 times) of this procedure will corrupt the flash itself and require a flash erase procedure that is built into more sophisticated JTAG software programmers (i.e., programmers that can erase and re-format the flash). This procedure assumes that the user has installed OCD Commander as directed in Section 3.4 above.

Step 1.
Open two i.MX GNU X-Tools Shells. One will run the OCDRemote program, while the other will be used to load the binary to the board.

Step 2.
Connect the provided USB cable to the usbDemon connector on the module board and the other end to the host computer USB connector. Connect the Ethernet and serial cables to the target expansion board and power on the board. Start your serial terminal (HyperTerminal, uCon or equivalent) to monitor board activity, which should be nothing at this point if disaster recovery is truly needed. If the board boots into MicroMonitor, then do not proceed from here; disconnect the usbDemon, go back to the previous section and perform the normal reflash procedure. Figure 3.8 shows the order in which connections to the target board should be made.

Figure 3.8 Order of Connections



Step 3.

Assuming a i.MXL target processor, issue the following command in the first i.MX GNU X-Tools Shell:

$ ocdremote –c ARM920T –d USB –a 0 –s 1 (enter)

(assuming '0' is the USB device number – use the 'usbDemon Finder' utility that accompanies the OCD Commander icon on the desktop if you are unsure of the device number). If targeting i.MX21, substitute 'ARM926EJ-S' for processor type.

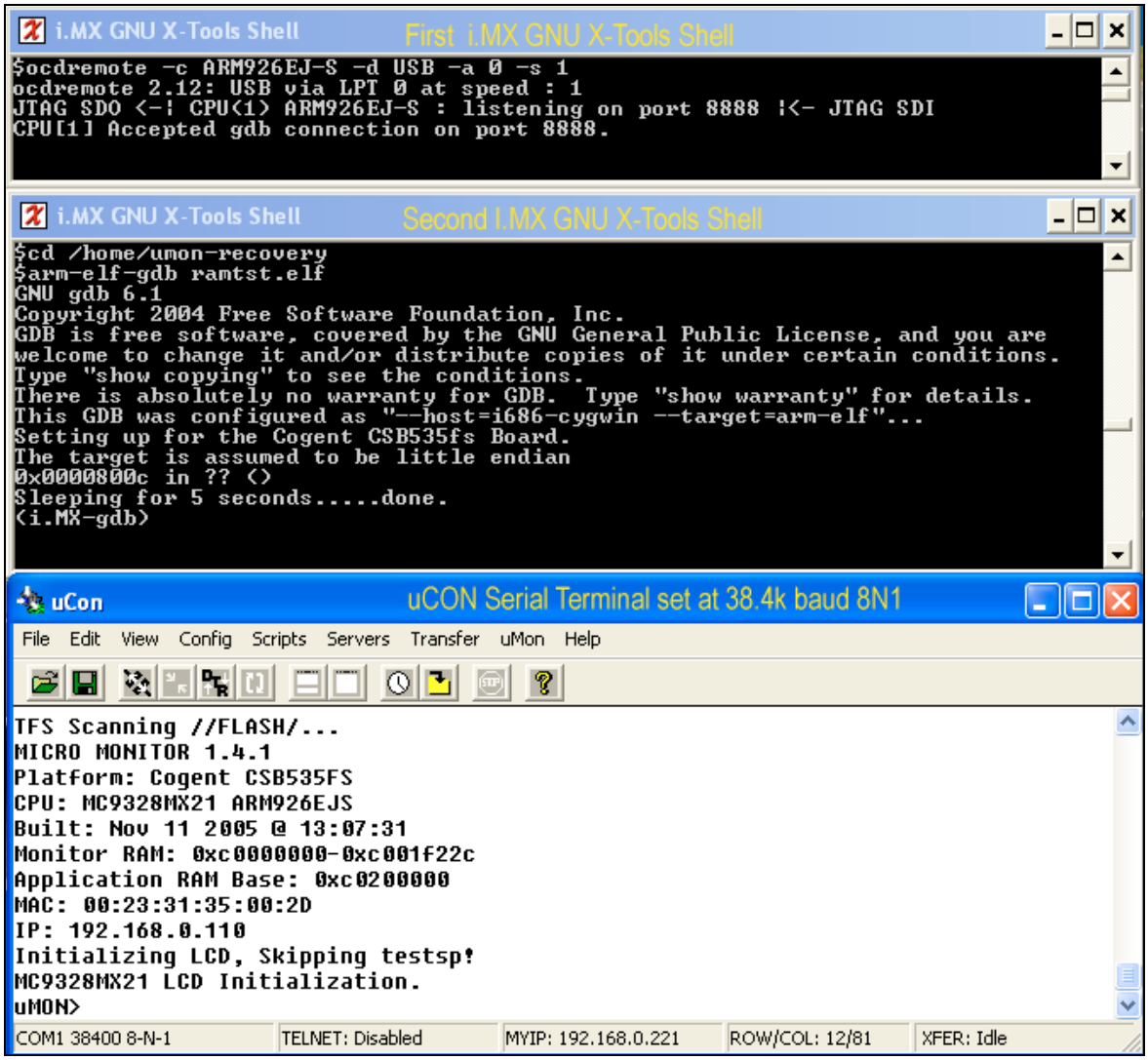
This shell should echo back the following:
"JTAG SDO <-| CPU(1) ARM920T : listening on port 8888 |<- JTAG SDI"
Each time a GDB session is open and closed, the OCDRemote utility will need to be recycled.

Issue the following commands in the second i.MX GNU X-Tools Shell:

$ cd /home/umon-recovery (enter)

$ arm-elf-gdb ramtst.elf (enter) (wait until the program has completely loaded into RAM and MicroMonitor autoboots as evident in the serial terminal session). Figure 3.9 shows the two i.MX GNU X-Tools shells and the uCon serial terminal. The first shell shows ocdremote operating; the second shows GDB loading ramtsts.elf image into RAM; and the third window shows uCon booting MicroMonitor from RAM.

Figure 3.9 Loading uMON into RAM


i.MX GNU X-Tools Shell
First i.MX GNU X-Tools Shell
$ocdremote -c ARM926EJ-S -d USB -a 0 -s 1
ocdremote 2.12: USB via LPT 0 at speed : 1
JTAG SDO <-| CPU(1) ARM926EJ-S : listening on port 8888 |<- JTAG SDI
CPU[1] Accepted gdb connection on port 8888.
i.MX GNU X-Tools Shell
Second i.MX GNU X-Tools Shell
$cd /home/umon-recovery
$arm-elf-gdb ramtst.elf
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i686-cygwin --target=arm-elf"...
Setting up for the Cogent CSB535fs Board.
The target is assumed to be little endian
0x0000800c in ?? ()
Sleeping for 5 seconds.....done.
(i.MX-gdb)
uCon
uCON Serial Terminal set at 38.4k baud 8N1
File Edit View Config Scripts Servers Transfer uMon Help
TFS Scanning //FLASH/...
MICRO MONITOR 1.4.1
Platform: Cogent CSB535FS
CPU: MC9328MX21 ARM926EJS
Built: Nov 11 2005 @ 13:07:31
Monitor RAM: 0xc0000000-0xc001f22c
Application RAM Base: 0xc0200000
MAC: 00:23:31:35:00:2D
IP: 192.168.0.110
Initializing LCD, Skipping testsp!
MC9328MX21 LCD Initialization.
uMON>
COM1 38400 8-N-1
TELNET: Disabled
MYIP: 192.168.0.221
ROW/COL: 12/81
XFER: Idle

Step 4.

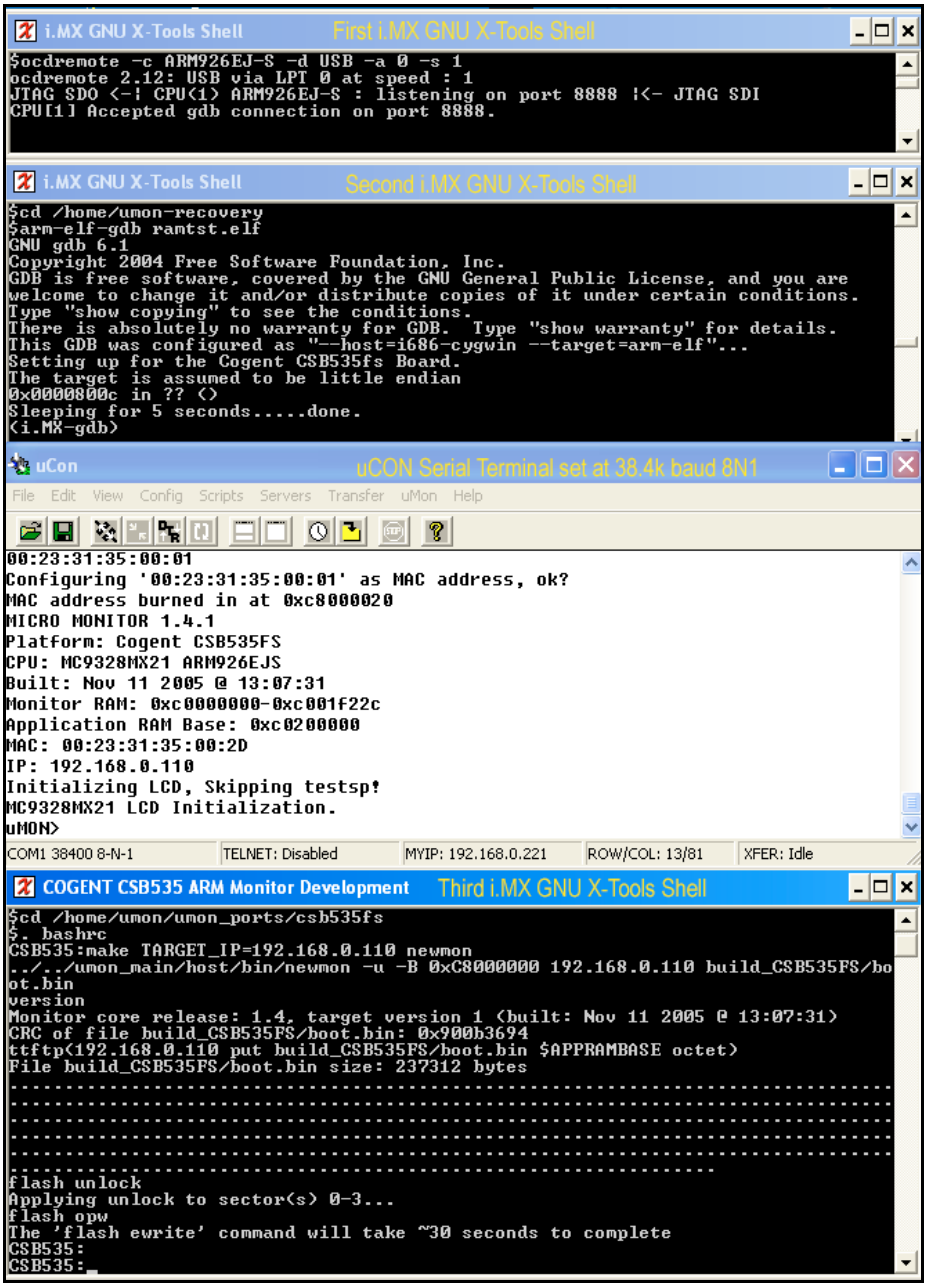
Open a third i.MX GNU X-Tools Shell window and issue the following commands:

```
$ cd /home/umon/umon_ports/csb536fs (enter)  (for MXL port – substitute ‘csb535fs’ for MX21)
$ . bashrc  (enter)  (dot space ‘bashrc’ sets dependency paths)
$ make TARGET_IP=xxx.xxx.xxx.xxx  newmon  (enter) (substitute actual board IP address)
```

Wait for approximately 30 seconds and power cycle the board to restart MicroMonitor. You will be prompted to input the last four digits of the MAC address. Use the last four digits on the label that is affixed to the board and separate every two digits with a ‘:’ colon. Confirm the new MAC address when prompted, and you are done. Click on ‘X’ to close out each i.MX GNU X-Tools shell window.

Figure 3.10 below shows three i.MX GNU X-Tools shells and a uCon serial terminal window. The first i.MX GNU X-Tools shell shows ocdremote in action; the second i.MX GNU X-Tools shell shows GDB connected to the target; the third i.MX GNU X-Tools shell shows the reflashing event; and last the uCon serial terminal shows a reboot with a MAC address prompt.

Figure 3.10 Reflashing uMON with uMON RAM Version


i.MX GNU X-Tools Shell
First i.MX GNU X-Tools Shell
$ocdremote -c ARM926EJ-S -d USB -a 0 -s 1
ocdremote 2.12: USB via LPT 0 at speed : 1
JTAG SDO <-¦ CPU(1) ARM926EJ-S : listening on port 8888 ¦<- JTAG SDI
CPU[1] Accepted gdb connection on port 8888.
i.MX GNU X-Tools Shell
Second i.MX GNU X-Tools Shell
$cd /home/umon-recovery
$arm-elf-gdb ramtst.elf
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i686-cygwin --target=arm-elf"...
Setting up for the Cogent CSB535fs Board.
The target is assumed to be little endian
0x0000800c in ?? ()
Sleeping for 5 seconds.....done.
(i.MX-gdb)
uCon
uCON Serial Terminal set at 38.4k baud 8N1
File Edit View Config Scripts Servers Transfer uMon Help
00:23:31:35:00:01
Configuring '00:23:31:35:00:01' as MAC address, ok?
MAC address burned in at 0xc8000020
MICRO MONITOR 1.4.1
Platform: Cogent CSB535FS
CPU: MC9328MX21 ARM926EJS
Built: Nov 11 2005 @ 13:07:31
Monitor RAM: 0xc0000000-0xc001f22c
Application RAM Base: 0xc0200000
MAC: 00:23:31:35:00:2D
IP: 192.168.0.110
Initializing LCD, Skipping testsp!
MC9328MX21 LCD Initialization.
uMON>
COM1 38400 8-N-1
TELNET: Disabled
MYIP: 192.168.0.221
ROW/COL: 13/81
XFER: Idle
COGENT CSB535 ARM Monitor Development
Third i.MX GNU X-Tools Shell
$cd /home/umon/umon_ports/csb535fs
$. bashrc
CSB535:make TARGET_IP=192.168.0.110 newmon
../../umon_main/host/bin/newmon -u -B 0xC8000000 192.168.0.110 build_CSB535FS/bo
ot.bin
version
Monitor core release: 1.4, target version 1 (built: Nov 11 2005 @ 13:07:31)
CRC of file build_CSB535FS/boot.bin: 0x900b3694
ttftp(192.168.0.110 put build_CSB535FS/boot.bin $APPRAMBASE octet)
File build_CSB535FS/boot.bin size: 237312 bytes
flash unlock
Applying unlock to sector(s) 0-3...
flash opw
The 'flash ewrite' command will take ~30 seconds to complete
CSB535:
CSB535:

3.8 MicroMonitor Familiarization and Tips

MicroMonitor is the boot ROM that comes with each Cogent development board. It is very useful in initializing hardware, starting serial and Ethernet services, and creating a tiny filesystem to store applications and the

embedded Linux startup binaries. Moreover, MicroMonitor facilitates loading the GX-Linux binaries using TFTP services over Ethernet or XModem through the serial port. We will focus on the TFTP (across Ethernet) transfer solution due to the size of the images and the time it takes to load binaries.

3.8.1 Manual TFTP Up-Load Procedure

Microcross ships each board pre-configured with the Linux script, 'startlinux', and two binaries: 'zImage' (kernel) and 'romfs.img' (root filesystem and Busybox utilities). To manually up-load files, follow the steps in this section. The upload address, <target-IP-address>, is the IP address of the target board.

Reload 'startlinux' Startup Script

To reload 'startlinux' script on the target board, you have three options to control your linux bootup:

- Load 'startlinux' for manual bootup (requires serial connection to use the uMON command line interface to execute 'startlinux'):

  $ cd <build-dir>/boot (enter)
  $ ttftp <target-IP-address> put startlinux startlinux,e (enter)

- Load 'startlinux' for autoboot with abort query:

  $ cd <build-dir>/boot (enter)
  $ ttftp <target-IP-address> put startlinux startlinux,eB (enter)

- Load 'startlinux' for autoboot without abort query – **warning** – there is no way to recover an error without disaster recovery efforts using a JTAG emulator to reload MicroMonitor into RAM and then write back to flash.

  $ cd <build-dir>/boot (enter)
  $ ttftp <target-IP-address> put startlinux startlinux,eb (enter)

Change 'startlinux' **Attribute Options**

There is another option to change 'startlinux' script flags without having to reload 'startlinux' again by using the serial uMON command line interface. Microcross', 'startlinux', is preloaded with the 'e' flag attribute set, which is the manual bootup setting. Two other boot options are available and can be set issuing one of the following commands:

1. Autoboot with abort query (case sensitive).

   uMON> tfs –feB cp startlinux startlinux (enter)

2. Autoboot without abort query (case sensitive) – **WARNING** – there is no way to recover from an error without disaster recovery efforts using a JTAG emulator to reload MicroMonitor into RAM and then write back to flash.

   uMON> tfs –feb cp startlinux startlinux (enter)

MicroMonitor training material is available by Microcross. Here is a link to more information / documentation: http://www.microcross.com/html/micromonitor.html

Reload ‘zImage’ Procedure

Similarly to the upload procedures above, you can add the ‘zImage’ – Linux kernel – to the TFS on uMON.

```
$ cd <directory-to-linux-zImage-binary>  (enter)
$ ttftp <target-IP-address> put zImage zImage  (enter)
```

Reload ‘romfs.img’ Procedure

Similarly to the upload procedures above, you can add the ‘romfs.img’ – root filesystem and Busybox utilities – to the TFS on uMON.

```
$ cd <build-directory>  (enter)
$ ttftp <target-IP-address> put romfs.img romfs.img  (enter)
```

3.8.2 Other Useful MicroMonitor Commands

uMON> tfs ls	List files in the Tiny File System
uMON> tfs clean	Defrag the Tiny File System without deleting files
uMON> tfs init	Deletes all files and defrags the Tiny File System – requires network setup
uMON> tfs rm <file-name>	Delete a specific file named <file-name>
uMON> help	Shows all of the available commands
uMON> help <command-name>	Shows specific help on command name
uMON> set	Shows all settings in memory
uMON> flash info	Shows all flash sectors and which ones are locked and erased
uMON> flash unlock x-y	Unlocks sectors x (lower bound) to y (upper bound)
uMON> flash erase x-y	Erase sectors x (lower bound) to y (upper bound)

Section 4. How to Use i.MX GNU X-Tools

4.1 Introduction

This section is intended to provide an abbreviated overview of using the GNU compiler and tools to generate executable programs for a target processor. Mastering the toolsuite and using it to develop embedded systems on a practical scale requires greater in-depth knowledge than can be covered here. If you are considering the GNU cross-tools for full-scale development, it is suggested that you acquire the manuals listed in the bibliography (see Bibliography). The ones having unrestricted reproduction rights are included in the docs directory on the i.MX GNU X-Tools CD-ROM. The O'Reilly book, *Programming with GNU Software*, by Mike Loukides and Andy Oram, is highly recommended, being complementary to this User Guide as well as a comprehensive guide to the GNU tools. The remaining texts provide more in-depth knowledge for specific toolsuite components as needed. In addition, Microcross offers a *GNU X-Tools Training Guide* as a complimentary product. It is based on an on-site three-day training course for beginning to intermediate level audiences. The course material is tailored to the i.MX GNU X-Tools for embedded development, and the format is geared to a self-paced training program.

4.2 Using the i.MX GNU X-Tools Command Line Tool (xtools)

The i.MX GNU X-Tools command line tool is a shell tool designed to provide several functions that facilitate convenient management of the i.MX GNU X-Tools cross-toolsuites from a command shell environment. For many developers, this method is more convenient than using a GUI. The following functions may be invoked from any 'Bash' Shell prompt:

1) xtools (enter) -- Execution of the 'xtools' command without arguments will show an abbreviated help screen that summarizes the command options described in this section. In addition, execution of an incomplete command (insufficient or incorrect arguments) will produce a help dialog for that command.

2) xtools <target-alias> -- This command starts a new bash shell having the appropriate environment variables and aliases initialized such that the cross-toolsuite components for the designated target are invoked by the familiar command names, in lieu of the native tool chain elements. For example, after starting a shell using the command line 'xtools arm-elf', the command GCC would actually start the compiler 'arm-elf-gcc'. This facility allows the command line user to execute the cross-tools using the same command names as used in the native shell environment. This remapping of tool command names remains in effect until the shell is exited via 'Ctrl-D' or the 'exit' command.

3) xtools status [path prefix] -- This command generates a summary screen which displays the installation status of each i.MX GNU X-Tools cross-toolsuite in terms of the types, counts, and locations of files associated with each toolsuite. The command displays the toolsuite status by target-alias vs. the population of files in the following format:

Column 0 Target Name (target-alias) for this toolsuite (i.e., arm-elf in our case)
Column 1 (/usr/bin/<target-alias-*) files
Column 2 (/usr/<target-alias>/bin/*) files
Column 3 (/usr/<target-alias>/lib/*) files
Column 4 (/usr/<target-alias>/include/*) files
Column 5 (/usr/lib/gcc-lib/<target-alias>/*) files
Column 6 (/usr/man/man1/<target-alias>-*) files
Column 7 Total file count for this toolsuite

The optional path prefix argument allows the utility to examine the installation state in which the user may have elected to manually install, or build/install toolsuite(s) into an alternate location (other than '/usr').

4) xtools install <target-alias> [archive-path] -- This command will install a complete cross-toolsuite from the distribution tarball on the specified archive path. All files are unarchived to the proper location (see above)

regardless of the current directory state, and subsequent invocation of a bash shell using the 'xtools <target-alias>' command will render the toolsuite ready to use. If the archive path is not provided, a default path is selected. The command 'xtools install' will display the default path, which is '/mnt/cdrom' unless it has been changed or overridden by the global environment variable '$XTOOLS_ARC_PATH'.

5) xtools remove <target-alias> [path prefix] -- This command allows the user to selectively remove a designated toolsuite from the system. In the unusual case where the toolsuite has been manually installed to some root prefix other than '/usr', the second option can be used to specify the root prefix (such as '/usr/bin') from which removal of all toolsuite files is desired. This command is most useful if you have performed a build from sources and subsequent install to an alternate root prefix path, and now desire to remove the installed tools.

6) xtools remove-all go [path prefix] -- The 'remove-all' command allows the user to summarily remove all toolsuites from the system. With the exception of the files '/usr/bin/xtools', '/usr/bin/xtools.rc', and '/etc/profile' (Cygwin only); this command affects a complete removal of i.MX GNU X-Tools toolsuite binaries from your system.

7) xtools install-libs <target-alias> [arcpath] -- This option allows an administrator to update or reinstall toolsuite libraries only. This feature is useful for selectively reinstalling libraries that have been over-written, or updating the libraries from a newer toolsuite version without updating the entire toolsuite. This option updates the following files:

/usr/<target-alias>/lib/*/libc.a
/usr/<target-alias>/lib/*/libstdc++.a

This step extracts the library files from a Microcross i.MX GNU X-Tools Toolsuite distribution CD-ROM, or filesystem image. The optional '[arcpath]' parameter may be used to locate the update files.

8) xtools install-hdrs <target-alias> [arcpath] – This option, like option 7, allows incremental updates to an installed toolsuite except that it updates the library headers files. It updates the following files:

/usr/<target-alias>/include/*/*.h

4.3 Invoking i.MX GNU X-Tools

To invoke i.MX GNU X-Tools from the command line in the shell, type 'xtools target-alias' and from that point on you do not need to type the alias at any point further. The target-alias will become part of the command line prefix (i.e., 'target-alias$'). To change or exit the target-alias mode, type 'exit' at the command line.

From Command Shell, type: $ xtools arm-elf (enter)
The prompt will now show: arm-elf$
To show the GCC version: arm-elf$ gcc -v (enter)

Issuing the following commands can be accomplished using the shortened version of tool name (i.e., gcc, as, ar, etc.), provided that you perform the 'xtools <target-alias>' command as shown above.

- *target-alias*-**gcc**
  Invokes all the necessary GNU compiler passes for the specific target processor toolsuite using (i.e., arm-elf, mips-elf, ppc-elf, etc.).
- *target-alias*-**cpp**
  Invokes the preprocessor that processes all of the header files and macros that the target requires.
- *target-alias*-**gcc**
  Invokes the C compiler that produces assembly language code from the processed C files.

- *target*-alias-**g++**
  Invokes the C++ compiler that produces assembly language code from the processed C++ files.
- *target*-alias-**gdb**
  Invokes the GNU Debugger with command line input (see Debugging with GDB in the PDF files on the i.MX GNU X-Tools CD.
- *target*-alias-**gdbtk**
  Invokes the GNU Debugger with a visual GUI interface called Visual GDB by Microcross (a.k.a. Insight).
- *target-alias*-**as**
  Invokes the GNU assembler that produces binary code from the assembly language code and puts it into an object file.
- *target-alias*-**ld**
  Invokes the linker that binds the code to addresses links the startup file and libraries to the object file and produces the executable binary image.

4.4 Using the i.MX GNU X-Tools Toolsuite

The first step to developing code with the i.MX GNU X-Tools toolsuite is creating and editing the source code. Microcross provides the Visual X-Tools IDE as companion product for code editing and project management; however, a programmer can use any text line editor to create source files and makefiles and use the i.MX GNU X-Tools as the production build environment. Free editors under Cygwin include VIM, an improved VI editor clone, and Xemacs, a powerful editor and project manager. Under Linux, the user also has a wide variety of choices including the Microcross Visual X-Tools IDE and GNU Xemacs edit (see docs in the Linux distribution).

4.4.1 Simple Example

Start a shell (xterm on Linux and i.MX GNU X-Tools Shell on Windows/Cygwin) and issue the command ‘xtools <target-alias>‘, where the target alias is arm-elf in our example.

The following simple example shows you quickly how easy it is to get a program compiled, linked and executed in an Instruction Set Simulator (ISS) and debug it using Visual GDB.

```
$ xtools arm-elf  (enter)
arm-elf$ cd /home/test  (enter)
```

In the last step, we assumed that the Microcross test directory either was installed in Cygwin’s /home/test or Linux’s /home/test.

```
arm-elf$ gcc –g –o pascal.x pascal.c   (enter)
arm-elf$ run pascal.x   (enter)
```

The program was compiled and linked in one step, and then run in the simulator.

```
Now issue the following command:
arm-elf$ gdbtk pascal.x   (enter)
```

The Visual GDB debugger is invoked and loads the target executable, pascal.x, for symbolic debugging.

Click on ‘File|Target Settings…’ and click on ‘Connection Target’ and select ‘Simulator’ and click ‘OK’.

Click on the ‘Run’ icon (left-most icon near menu) and begin single stepping. To see the program output, click on ‘View|Console’.

When finished, click on ‘File|Exit’.

This concludes the simple example showing how easy it is to compile and run a program with i.MX GNU X-Tools. The i.MX GNU X-Tools toolsuite is very powerful and can build any conceivable program a developer can code. Now, we need to go into more detail on how the i.MX GNU X-Tools work.

4.4.2 i.MX GNU X-Tools Toolsuite Description

The GCC program is actually a control program that executes the compiler components to produce the desired output, which is usually a compiled and linked executable program image. By manipulating the many GCC options and controlling the input file types, the functions of GCC are greatly expanded. The GCC, however, is but a single component. It is actually a control program that calls other components that perform separate steps to create an executable binary. The components are described as follows and show graphically in Figure 4.1:

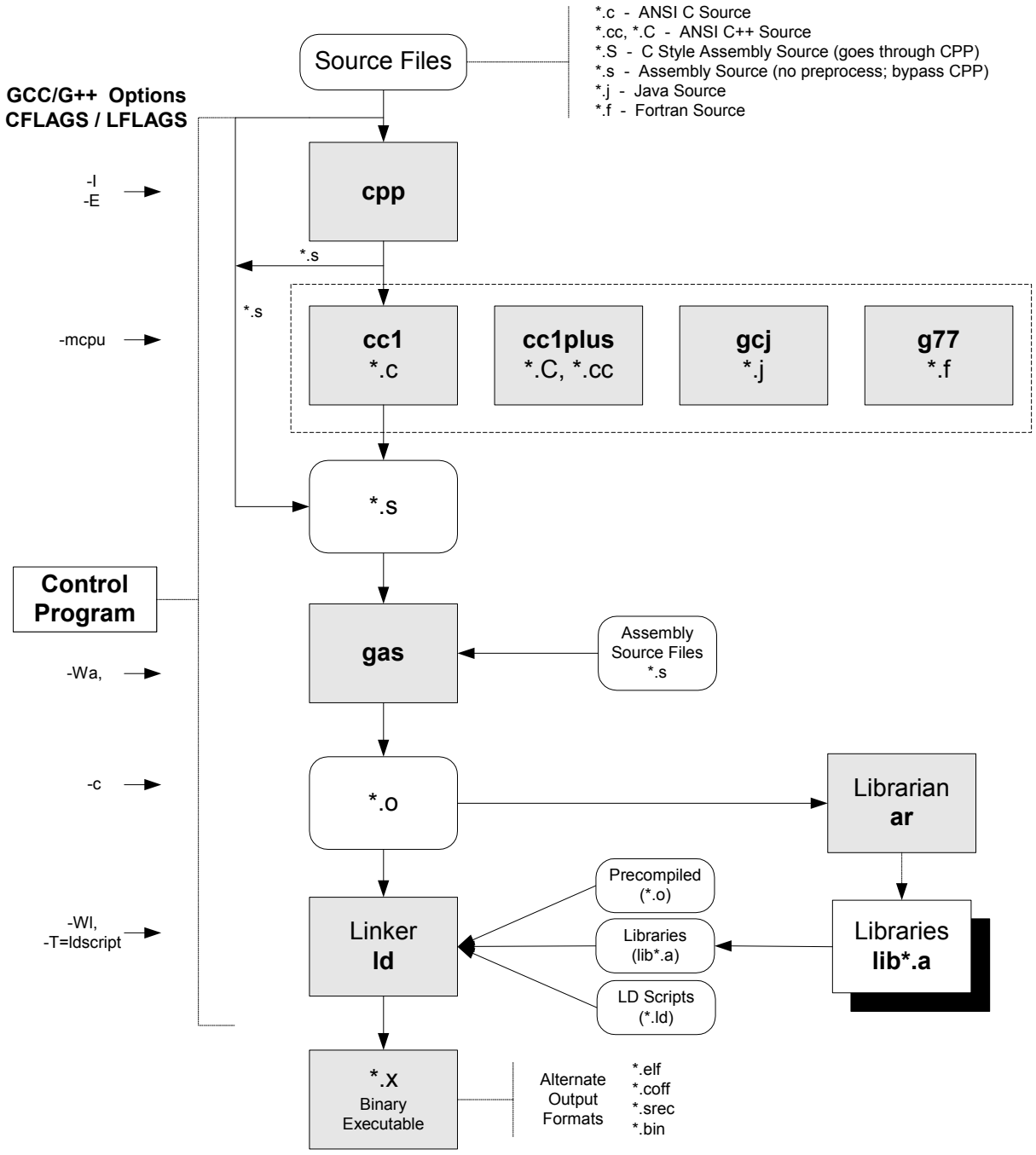
preprocessor
Performed by cpp, which is invoked by GCC, the preprocessor resolves directives like '#define', '#include', and '#if'. Preprocessing establishes the type of source code to process.

compiler
Performed by GCC, the compiler pass, which produces assembly language from the input files, and passes the assembly source directly to the assembler phase.

assembler
Performed from GCC by as, the GNU assembler. It takes the assembly language as input and produces object files with '.o' extensions. The assembler output is relocatable object code ('.o').

linker
Performed by ld, the GNU linker. Linking completes the compilation process, combining all object files (newly compiled, and those specified as input) into an executable file. This step completes the final stage, where the '.o' modules are placed in their proper places in the executable file. Library functions that the program refers to are also placed in the file. GCC performs this task by internally invoking the linker. GCC also cleans up by deleting any object files that it created from source files; however, it does not cleanup any pre-existing object files that you specified on the command line. GCC normally invokes all of these compilation steps when converting a C source program into an executable. By using command line options for GCC, these steps may be invoked separately or in some combinations. This provides some flexibility when building large programs, or using assembly language sources, or debugging.

Figure 4.1. i.MX GNU X-Tools Flow Diagram



4.4.3 Control Program (GCC)

Exactly how the GCC processes any file depends on the file's name. The control program strips the initial part of the name, and then determines how to process the file on the basis of the filename's extension. In each case, the control program passes the file to the appropriate program for preprocessing, compilation, or assembly, and it links

all resulting object modules together to produce an executable file. The following table shows how the control program recognizes different file types:

Table 4.1. Controlling Compilation / Linking

Input File	Interpretation	Action
file.c	C source*	Preprocessed and compiled by GCC
file.C	C++ source **	Preprocessed by GCC and compiled by g++
file.cpp	C++ source *	Preprocessed by GCC and compiled by g++
file.CPP	C++ source **	Preprocessed by GCC and compiled by g++
file.cc	C++ source *	Preprocessed by GCC and compiled by g++
file.cxx	C++ source *	Preprocessed by GCC and compiled by g++
file.h	C or C++ header file	Precompiled header
file.s	Assembly language source*	Assembly by as
file.S	Assembly language source**	Preprocessed and assembled by as
file.o	Compiled object module*	Passed to ld
file.a	Object module library*	Passed to ld

* Lower case file extension.
** Upper case file extension.

All other files, together with options that GCC does not recognize, are passed to ld, the linker – either the 'native' linker supplied by your vendor, or the GNU linker. As a result, almost all linker options are available directly through GCC.

In the commands shown earlier in this Section, the '–l' and '–L' options were actually passed to the linker. In general, GCC passes on unrecognized options to the linker, so you can specify linker options without having to invoke the linker separately.

If you're writing C++ code, you can use the g++ command instead of GCC. You're actually getting the same compiler; however, when it is invoked as g++, the compiler expects C++ instead of vanilla C source code. In addition, g++ uses different default libraries.

If you want either to preserve the output of GCC at some intermediate stage for debugging purposes or to manipulate the code directly, you can do so. Here is how each stage works:

preprocessing

A '–E' option in GCC sends the preprocessed code to the standard output, instead of compiling the program.

compilation

To save the assembly language output, run GCC with the '–S' option. This produces files whose names end with '.s' in place of the source file's '.c'.

assembly

As we have seen, running with '–c' produces object files whose names end with '.o'.

We have just finished a long discussion of the many kinds of input that the compiler takes and the different kinds of output that it can provide. Pictures are not always worth a thousand words, particularly when it comes to summarizing a lot of disparate information. But it may help you to remember this information if you view the compiler as a kind of "machine" with different inputs and outputs, as outlined earlier.

The input file's name determines where it goes into the machine. The '.c', '.S', '.cc', '.cpp' and '.C' files go to the preprocessor, '.i' and '.ii' files go straight to the compiler, and so on. Compilation options determine which stage of the machine produces output (i.e., how many stages of the machine you run). The '–S' means that you stop after the assembler and the output filename ends with '.s'. If you keep this picture in mind, the compiler's machinations will not seem so strange; you will stop seeing preprocessing, compilation, assembly, and linking as separate steps and come to see compilation as one big assembly line, for which GCC is the production manager.

The following tools listed in Tables 4.2, 4.3, and 4.4 are the main tools for developing projects with a i.MX GNU X-Tools Toolsuite.

Table 4.2. i.MX GNU X-Tools Compiler and Development Tools

Tool	Description
as	GNU assembler
cpp	C preprocessor
gcc	Optimizing ANSI compliant GNU Compiler Collection (controller of all compilation)
gdb	GNU debugger for source and assembly debugging with command line
gdbtk	Debugger using a graphical user interface called Visual GDB (a.k.a. Insight)
g++	Optimizing ISO tracking GNU C++ compiler
gasp	GNU assembler preprocessor
ld	GNU Linker

Table 4.3. i.MX GNU X-Tools Libraries

Tool	Description
libc	Non-restricted ANSI C runtime library *(Newlib for cross-development*)
libgloss	Support library for embedded targets (*board support for cross-development*); (deprecated) will be replaced by a future product
libm	Non-restricted C math subroutine library *(Newlib for cross-development*)
libstdc++	GPL C++ class library, implementing the ISO 14882 Standard C++ library

Table 4.4. i.MX GNU X-Tools Binary Utilities

Tool	Utility Description	Target Dependent
addr2line	Converts addresses into file names and line numbers	yes
ar	Creates, modifies and extracts from object code archives	yes
diff diff3 Sdiff	Comparison tools for text files	no
make	Compilation control program	no
nm	Lists symbols from object files	yes
objcopy	Copies and translates object files	yes
objdump	Displays information from object files	yes
patch	Installation tool for source fixes	no
ranlib	Generates index to archive contents	yes
readelf	Displays information about ELF format object files	no
run	Standalone simulator	yes
size	Lists file section sizes and total sizes	yes
strings	Lists printable strings from files	yes
strip	Discards symbols	yes

The Adobe Acrobat (PDF) files listed in Table 4.5 are available on the i.MX GNU X-Tools or Cygwin (docs directory) CD-ROM for reference or hard-copy duplication.

Table 4.5. PDF Documentation on CD-ROM

No.	File Name	Description	Pages
1	Bench++.pdf	By Joe Orost, Bench++ is designed to measure the performance of the code generated by C++ compilers, not hardware performance.	15
2	ddd.pdf	Debugging with DDD (Data Display Debugger), v3.3.9, 15 January 2004	234
3	Infoman.pdf	Reading GNU On-Line Documentation	36
4	Make.pdf	How to use GNU Make	166
5	Man-pages.pdf	A Concise Reference Document for all of the Tools in Unix Man Page Format	156
6	MC-Auxiliary-Tools.pdf	Using as Using binutils Using cygwin Using info	340
7	MC-Compiler-Tools.pdf	Using GNU CC Using the C Preprocessor	412
8	MC-Debugging-Tools.pdf	Debugging with GDB Insight, the GNUPro Debugger GUI Interface	260
9	MC-Embedded-Systems.pdf	GNU Tools for Embedded Systems…Everything you want to know about using GNU tools with embedded target processors.	530
10	MC-Libraries.pdf	GNU C Library GNU Math Library GNU C++ iostream Library	294
11	MC-Development-Tools.pdf	Using ld Using make Using diff & patch	350
12	MC-Utilities.pdf	Using as Using ld Using binutils Using make Using diff & patch Using info	682

4.5 Controlling the Tools Using the GCC

The GCC (GNU Compiler Collection) control program can be run like the following, producing an executable file from a number of C or C++ source files. Both examples assume that the target toolsuite is installed prior to executing the build commands. The examples shown are with GCC, which controls the compilation of C, C++, and assembly files. Also, the user must name the C files with the '.c' extension and C++ files with '.C' , '.cpp' or '.cc' extensions.

$ target-alias-gcc –options –o <program.out.name> <first.c> <second.c> <third.c> <fourth.c> … (enter)

Or

$ xtools <target-alias> (enter)
<target-alias>$ gcc –options –o <program.out.name> <first.c> <second.c> … (enter)

Or

$ xtools <target-alias> (enter)
<target-alias>$ gcc - options one_or_more_source_files.c -o program.out.name (enter)

Or

<target-alias>$ gcc - options -c source_file.c (enter)
<target-alias>$ gcc - options file1.o . . . fileN.o -o program (enter)

All alternatives actually look the same to the compiler parts. The program executed under the name of GCC is just a front that handles options and temporary files and calls the real compiler parts: cpp, the C preprocessor. It takes care of preprocessor directives, such as include files and macro expansions. It also removes comments. The result is a file with the C code, lots of white space and some line-numbering directives that the compiler core can use in warning and error messages.

The '–o' filter argument tells GCC to name the executable file 'program'. If you don't specify an '–o' argument, GCC chooses the default name 'a.out', which is not particularly informative (and would cause multiple executables to overwrite each other). So, most programmers use the '–o' argument to name the program.

Since the inputs to the last example above are all object files, no compilation or assembly is required: GCC always invokes the linker. Using GCC to invoke the linker is preferable to using ld separately, because GCC ensures that the program is linked with the correct libraries and initialization routines.

4.5.1 GCC Options Commonly Used

-c
Compile or assemble the source files, but do not link. The linking stage simply is not done. The ultimate output is in the form of an object file for each source file.

-o *file*
The '-o' and not '-c' tells GCC to create a linked executable with a user defined 'filename' as its output file. Not specifying '-c' or '-o' will tell GCC to create a default executable named 'a.out' for all cross-compilers and 'a.exe' for the native Cygwin compiler.

-D
One, the '–D' option, acts like '#define' in the source code: it sets the value of a symbol.

<target-alias>$ gcc –c –D first=\"info\" –D second example.c (enter)

The first '–D' option sets 'first' to the string 'info\' (because of the backslashes, the quotation marks actually become part of the symbol's definition). This can be useful for controlling which file a program opens. The second '–D' option defines the 'second' symbol. It happens to set it to the value 1, the default, but you probably don't care; your program just uses an '#ifdef' directive to check whether it's set.

-E
Stop after the preprocessing stage; do not run the compiler proper. The output is in the form of preprocessed source code, which is sent to the standard output. Input files which don't require preprocessing are ignored.

-S

Stop after the stage of compilation proper; do not assemble. The output is in the form of an assembler code file for each non-assembler input file specified.

By default, the assembler file name for a source file is made by replacing the suffix .c, .i, etc., with .s.

Input files that don't require compilation are ignored.

-v

Print (on standard error output) the commands executed to run the stages of compilation. Also print the version number of the compiler driver program and of the preprocessor and the compiler proper.

-###

Like -v except the commands are not executed and all command arguments are quoted. This is useful for shell scripts to capture the driver-generated command lines.

-pipe

Use pipes rather than temporary files for communication between the various stages of compilation. This fails to work on some systems where the assembler is unable to read from a pipe; but the GNU assembler has no trouble.

--help

Print (on the standard output) a description of the command line options understood by gcc. If the '-v' option is also specified then '—help' will also be passed on to the various processes invoked by gcc, so that they can display the command line options they accept. If the '–Wextra' option is also specified then command line options which have no documentation associated with them will also be displayed.

--target-help

Print (on the standard output) a description of target specific command line options for each tool.

--version

Display the version number and copyrights of the invoked GCC.

-Uname

Some symbols – usually those identifying the type of computer system you're compiling on – are automatically defined by the compiler. If you want to suppress one of these symbols, use the '–Uname' option. This is equivalent to putting '#undef name' at the beginning of each source file.

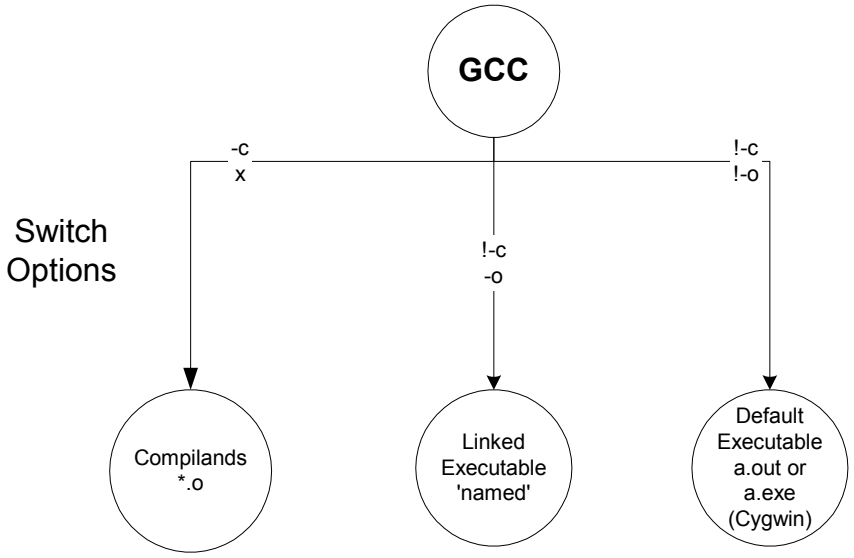
-I

If you have an include file in a non-standard directory, specify this directory in a '–I' option. For example, suppose you have two directories for code, '/usr/src' for source files and '/usr/headers' for header files. While you're compiling in the '/usr/src' directory, you can tell GCC where to find the header files through the command:

<target-alias>$ gcc –I../headers example.c

Figure 4.2 below summarizes the three primary modes of GCC with switch actions.

Figure 4.2. Three Primary Modes of GCC



4.6 Controlling Linker from GCC

Another common compilation option is '–l', which specifies a library and 'L', which specifies the path to a non-standard library. Important, '-l' must be specified at the end of the command after the source/object filenames, while all other options are specified before the source/object filenames. The examples shown are with GCC, but the user can substitute g++ in place of gcc along with the appropriate C++ files and libraries needing linked. Here's a typical command; the inputs are object files, so the command just runs the linker.

<target-alias>$ gcc –o example –L/src/local/lib main.o first.o –lm (enter)

This command tells GCC to look for libraries first in '/src/local/lib' path, then in the standard location. If someone has put a local version of a library in '/src/local/lib', that version is used in preference to the standard version in '/usr/lib'. The '-lm' specifies the library name -- actual library name is 'libm.a'. The prefix and suffix to the library name can be dropped on all library names. The naming convention for GNU libraries is lib<name-of-lib>.a (static libs) and lib-<name-of-lib>.so (shared libs).

Unix / Linux linkers search libraries in the order in which they occur on the command line and only resolve the references that are outstanding at the time the library is searched. Therefore, the order of libraries and object modules on the command line can be critical (this is why '–l' options come after the filenames). Consider the command:

<target-alias>$ gcc –l mine first.c (enter)

This command searches for the library file 'lib**mine**.a' to resolve any function references needed for linking; however, the linker has not yet processed the object module for 'file4.o' (created by the GCC command and normally deleted if compilation and linking are successful). Therefore, there are no outstanding function references, and the library search has no effect. If the program needs this library, the GCC command produces 'Undefined symbol' messages during the loading phase, and the linker does not produce an executable file. To perform this compilation correctly, enter the command:

<target-alias>$ gcc first.c –lmine

Now the loader searches the library after processing ‘first.c’ and is able to resolve any references requiring this library.

When compiling a C program, you often don’t need to list any libraries explicitly on the command line. The GCC automatically searches the system call library, I/O library, and run-time initialization routines. If you use any math routines, you’ll need to search for the math library (‘-lm’); if you’re compiling C++ code, you need to include the C++ libraries (‘-lstdc++’).

4.7 Compilation Options

The following sections discuss other important options that are available with the GCC. There are many, many other options – perhaps several hundred all-together – that control various details of compilation and optimization. The chances are that you will never need these, but it won’t hurt to familiarize yourself with the complete documentation for GCC. Complete coverage of GCC options is provided in *Using and Porting GNU CC* [reference 20].

4.7.1 Displaying compiler behavior

The ‘–v’ (verbose) option prints the compiler’s version number and complete details about how each pass is executed. This option is particularly useful for finding out exactly which options your program is being linked with.

4.7.2 C Language Options

We’ll start by listing a few options for controlling the warning messages that GCC produces. There are many options for controlling warnings; it’s possible to request (or inhibit) many warning messages on a per-message basis. We’re not sure that’s really useful; we’ll limit ourselves to a few options that control large group of messages:

-w
Suppress all warning messages.

-W
Produce some additional warning messages about legal (but questionable) coding practices.

-Wall
Produce even more warning messages about questionable coding practices.

-Wtraditional
Produce warning messages about code that is legal in both the ‘Kernighan and Ritchie’ and ANSI definitions of the C language, but behaves differently in each case.

-Werror
Make all warnings into errors; that is, don’t attempt to produce an object file if a warning has occurred.

Now, we’ll discuss how to control various features of the C and C++ languages. There are basically three options to worry about: ‘-traditional’, ‘-ansi’, and ‘-pedantic’. In most cases, it’s fairly easy to tell which you want. Older C code – code that pedants the standard – should be compiled with ‘–traditional’. Newer code that has been written to conform to the ANSI standard should be compiled with ‘–ansi’. Either option can accept prototypes, where you specify the arguments on the same line as the function name, as in ‘func (char *arg)’.

Note that the ANSI C standard and “traditional” (Kernighan and Ritchie) C both define the behavior of the preprocessor – either explicitly or implicitly. Therefore, the options listed below affect both cpp (C preprocessor) and GCC:

-traditional

Supports the traditional C language, including lots of questionable, but common, practices. The traditional option also supports all of the FSF’s extensions to the C language.

-ansi

Supports the ANSI C standard, though somewhat loosely. The FSF’s extensions are recognized, except for a few that are incompatible with the ANSI standard. Thus, ANSI programs compile correctly, but the compiler doesn’t try too hard to reject non-conformant programs, or programs using non-ANSI features.

-pedantic

Issues all the warning messages that are required by the ANSI C standard. Forbids the use of all the FSF extensions to the C language and considers the use of such extensions errors. As the GCC manual points out, ‘-pedantic’ is not a complete check for ANSI conformance, it only issues errors that are required by the ANSI standard.

4.7.3 Preprocessor options

The following set of options control the cpp preprocessor from the command line:

-M

Read the source files; figure out which files they include, and output lists of dependences for make. There is one dependency list for each source file. The dependency lists are sent to standard output, and compilation doesn’t proceed past preprocessing (i.e., ‘-M’ implies ‘–E’). This option can make it much easier to generate correct makefiles.

-C

The preprocessor normally deletes all comments from the program. With ‘–C’, it doesn’t. This flag may be useful in conjunction with ‘–E’ when you are trying to make sure that the preprocessor is doing what you intended. In such cases, leaving you comments in may be handy. The ‘–C’ option doesn’t automatically imply ‘–E’, but GCC won’t let you use ‘–C’ on the command line unless ‘–E’ is also present.

4.7.4 Options to Specify Libraries, Paths and Startup Files

The following options are common for embedded developers, but not in native environments:

-nostartfiles

Don’t use the standard system startup files when linking. Normally the ‘crt0.o’ file gets linked in as the standard start file; however, most embedded developers will need to replace the standard start file with a custom start file. When used with GCC, add the correct option ‘–Wl,-nostartfiles’ and add the custom startfile with appropriate entry symbol to the linker script (e.g., ENTRY(_start)). To learn more about how to create linker scripts and startfiles, read the i.MX GNU X-Tools Training Guide or the Microcross Visual X-Tools User Guide.

-nodefaultlibs

Do not use the standard system libraries when linking. Only the libraries you specify will be passed to the linker. The standard startup files and used normally, unless ‘-nostartfiles’ is used. The compiler may generate calls to memcmp, memset, and memcpy for System V (and ANSI C) environments or to bcopy and bzero for BSD

environments. These entries are usually resolved by entries in libc. These entry points should be supplied through some other mechanism when this option is specified.

-nostdlib

When linking, this option tells the linker do not use the standard libraries or startup files. This option is useful when you want to provide your own libraries, overriding the default libraries and use your own custom startup files. When used with GCC, add the correct option '–Wl,-nostdlib'. If you plan on augmenting the standard libraries with your own libraries, then do not use this option. Use a 'GROUP' and 'SEARCH_DIR' statement with appropriate arguments in the linker script – see the Visual X-Tools IDE User Guide or the i.MX GNU X-Tools Training Guide for examples on how to create linker scripts and startfiles.

-nostdinc

Do not search the standard system directories for header files. Only the directories you have specified with '-I' options and the current project directory are searched. By using both '-nostdinc' and '-I-' you can limit the include file search path to only those directories you specify explicitly.

-static

Link only to static libraries, not shared libraries. When used with GCC, add the correct option '–Wl,-static'. In i.MX GNU X-Tools, static link is the default setting.

-shared

If shared libraries are available, use them wherever possible, rather than static libraries. When used with GCC, add the correct option '–Wl,-shared'.

-I*dir*

Add the directory *dir* to the head of the list of directories to be searched for header files. This can be used to override a system header file, substituting your own version, since these directories are searched before the system header file directories. If you use more than one '-I' option, the directories are scanned in left-to-right order; the standard system directories come after.

-I-

Any directories you specify with '-I' options before the '-I-' option are searched only for the case of '#include "file"'; they are not searched for '#include <file>'. If additional directories are specified with '-I' options after '-I-', these directories are searched for all '#include' directives. Ordinarily all '-I' directories are used this way. In addition, the '-I-' option inhibits the use of the current directory (where the current input file came from) as the first search directory for '#include "file"'. There is no way to override this effect of '-I-'. With '-I.' you can specify searching the directory that was current when the compiler was invoked. That is not exactly the same as what the preprocessor does by default, but it is often satisfactory. '-I-' does not inhibit the use of the standard system directories for header files. Thus, '-I-' and '-nostdinc' are independent.

-L*dir*

Add directory *dir* to the list of directories to be searched for '-I'.

-B*prefix*

This option specifies where to find the executables, libraries, include files, and data files of the compiler itself. The compiler driver program runs one or more of the subprograms 'cpp', 'cc1', 'as', and 'ld'. It tries prefix as a prefix for each program it tries to run, both with and without 'machine/version/'. For each subprogram to be run, the compiler driver first tries the '-B' prefix, if any. If the name is not found, or if '-B' was not specified, the driver tries two standard prefixes, which are '/usr/lib/gcc-lib/' and '/usr/local/lib/gcc-lib'. If neither of those results in a file name that is found, the unmodified program name is searched for using the directories specified in your 'PATH' environment variable. '-B' prefixes that effectively specify directory names also apply to libraries in the linker, because the compiler translates these options into '-L' options for the linker. They also apply to includes files in the preprocessor, because the compiler translates these options into '-isystem' options for the preprocessor. In this case, the compiler appends 'include' to the prefix. The run-time support library file 'libgcc.a' can also be searched for using the '-B' prefix, if needed.

-specs=*file*

Process *file* after the compiler reads in the standard 'specs' file, in order to override the defaults that the 'gcc' driver program uses when determining what switches to pass to 'cc1', 'cc1plus', 'as', 'ld', etc. More than one '-specs=' file can be specified on the command line, and they are processed in order, from left to right.

4.7.5 Debugging and Profiling Options

These options request the compiler to create additional code and an expanded symbol table for the various profilers and debuggers (dbx, prof, gprof, and the branch count profiler). They are extremely helpful for debugging and tuning code under development, but should not be used for production release versions of your program.

-p

Link the program for profiling with prof. When you execute a program compiled with this option, it produces a file named 'mon.out' that contains program execution statistics. The profiler prof reads this file and produces a table describing your program's execution.

-pg

Link the program for profiling with gprof. Executing a program compiled with this option produces a file named gmon.out that includes execution statistics. The profiler gprof reads this file and produces detailed information about your program's execution. For example, the following command compiles the file 'program.f', generating code for profiling with gprof:

<target-alias>$ gcc –pg program.f (enter)

-g

Generate a symbol table for debugging. The '–g' option enables debugging with the GNU debugger, GDB or Visual GDB. The symbols used with standard GDB are called stabs. When compiling with Microcross' cross-compilers, set the '–g' option to produce the stabs symbols; you may use '–gstabs' as a verbose switch if you like. Other symbol options include '–gcoff', '–gdwarf' and '–gdwarf2'; the '–gcoff' symbols are not supported in Microcross GNU cross-compilers; however, '-gdwarf' and '-gdwarf2' are supported in most target CPU toolsuites.

IMPORTANT NOTES: To perform assembly debugging of straight assembly language files (.s or .S files), you must use '–g' or '–gstabs' because the assembler will not generate any other symbol formats. The C/C++ compiler, however, will generate appropriate symbols for either '-gdwarf' or '-gdwarf2' in addition to stabs. When compiling with any debug option, do not use optimization options. You cannot optimize code and debug it correctly using the GNU tools.

4.7.6 Optimization

i.MX GNU X-Tools GCC incorporates a sophisticated optimizing compiler; and for most target systems, it usually generates faster code than with the native compiler. Any '-Ox' optimization can be used with any '-fx' option (e.g., gcc -Os –funroll-loops –ffast-math –o outfile.x infile.c). Here are the most commonly used compilation options:

-O0
No optimization. This is the default. With optimization turned off, GCC tries to generate code that is easy to debug; you can set a breakpoint between any two statements and modify variables, and the program will behave exactly as it should. The GCC also tries to generate code quickly.

-O1
The compiler tries to moderately to reduce both the size of the compiled code and the execution time. Compilation is slower than with '–O0' and requires more memory.

-O2
Enables more optimizations than '–O1'. Compilation time is even slower; the resulting code should be even smaller and faster.

-O3
Enables more optimizations than '–O2'. Try hardest of all to produce fast assembly code. Note that the emphasis is on fast: the resulting code may take much more room in memory because certain functions may be placed in-line and loops may be unrolled (as if each iteration in the for loop were written out independently).

-Os
Try to produce code that is small. The emphasis is on size, not speed. This option uses many of the same optimization algorithms as '–O2', but with a different emphasis.

-ffast-math
Make floating-point arithmetic optimizations that violate the ANSI or IEEE standards. Code compiled with this option may yield incorrect results, but it will be slightly faster. Be sure to test your program thoroughly.

-finline-functions
Expand all "simple functions" into their callers. The compiler gets to decide whether any function is "simple" or not. Inline expansion is a two-edged sword; it can make a program faster (by eliminating calling overhead) or slower (by making instruction cache performance worse).

-fno-inline
Inhibit all inlining, even inlining that is requested by the inline keyword in the source code. The GCC performs inlining according to statements in the source code with both '–O1' and '–O2'; the keyword is ignored if optimization is not in effect.

-funroll-loops
On some CPUs, loop unrolling can be a very important optimization. It minimizes loop overhead and creates many opportunities for further optimizations. With the '–funroll-loops' option, GCC unrolls all loops that have a fixed iteration count known at the time of compilation. Loop unrolling is another risky optimization; it can improve performance, but it can also penalize performance significantly: the object file becomes larger, which can significantly hurt cache performance. Compile time also increases.

4.7.7 Passing Options to the Assembler or Linker

The GCC allows you to pass options directly to the assembler or linker when they are involved:

-Wa,options
Pass the 'options' to the assembler

-Wl,options
Pass the 'options' to the linker

In both cases, the 'option-list' is just a list of options recognized by the assembler or the linker. There must not be any spaces in the list; options in the list are separated by commas.

Here is an example that is both instructive and useful: producing a listing of the assembly language generated, together with C source listings. To do this, we need to pass the '–alh' options to the assembler (generate listings of assembly code and high-level source); we also need to pass the '–L' option to the assembler (retain local labels). And we need GCC's '–g' option (generate additional symbols for debugging; the additional symbols tell the assembler where to find the source code). The resulting command looks like this:

<target-alias>$ gcc –c –g –Wa,-alh,-L source.c (enter)

Listings that include both assembly and source code are interesting from two standpoints. You may want to see how your code has been compiled; this is instructive, whether or not you are optimizing and even if you are not interested in assembly level debugging. What is more important, though, is that you can generate a C / assembly listing for optimized code. This can be very helpful for debugging under optimization. The big problem with debugging optimized code is that there is no longer a simple mapping from your source code into assembly language; therefore, the debugger cannot single step and perform symbolic debugging. With a listing, you can find out exactly what the compiler did to your code and get a much better idea of what the code is doing.

4.8 Using the GNU Assembler

The GNU assembler is really many assemblers folded into one (or many different programs with the same name, depending on how you look at it). You can usually ignore the assembler; the compiler invokes it automatically and is usually able to specify everything the assembler needs to know about your environment. In rare cases, you may need to ask for an assembly option explicitly; in these cases, you will need to run the assembler, as, a separate program or use GCC's '–Wa' option to pass additional options to the assembler. The assembler arguments must be separated from each other (and the `-Wa') by commas. For example:

<target-alias>$ gcc -c -g -O -Wa,-alh,-L file.c (enter)

This above example emits a listing to standard output with high-level and assembly source.

Usually you do not need to use this `-Wa' mechanism, since many compiler command-line options are automatically passed to the assembler by the compiler. You can call the GNU compiler driver with the `-v' option to see precisely what options it passes to each compilation pass, including the assembler.

Next, we discuss what the assembler does. We will not discuss the assembly language itself. The FSF's documentation explains the general syntax of assembly language, but refers you to the vendor's architecture manual for CPU dependent details: overall architecture, instruction set, etc..

The assembler takes a program written in an assembly language and produces an object module. By convention, assembly language programs have the extension '*.s*'. If no errors occur during assembly and if the object module contains no references to external (imported) symbols, the assembler makes the file executable and names it 'a.out'. If the object module includes references to external symbols, 'a.out' is not an executable. The linker is able to link this object module with other modules to produce an executable program.

To invoke the assembler, enter the command:

<target-alias>$ as -options <list-of-source-files> (enter)

Where 'list-of-options' is a series of assembly options and 'list-of-source-files' is one or more assembly language files (.s) -- .S (capital S) requires GCC to direct the source file to the preprocessor and then on to the assembler . Unlike most Unix assemblers, the GNU assembler can work on several files at a time.

The assembler has many options; most of them are architecture-specific and are used to describe the target processor more precisely. These options will be important to you if you are cross-compiling; check the FSF's manual for more details. Each appendix in this user guide briefly discusses the target dependent compiler and assembler options.

The following are a set of options and controls that are generally useful for invoking all GNU assemblers:

Common assembler switch options to all Targets

as	[-a[cdhlns][=file]] [-D] [--defsym sym=value]
	[-f] [--gstabs] [--help] [-I dir][-J][-K][-L]
	[--keep-locals] [-o objfile] [-R] [--statistics]
	[-v] [-version] [--version] [-W] [-w] [-x] [-Z]
	[-mbig-endian | -mlittle-endian] (if implemented)
	[-mfpa10 | -mfpa11 | -mfpe-old | -mno-fpu] (if implemented)
	[-EB | -EL] (if implemented)
	[-O]
	[-O | -n | -N]
	[-b] [-no-relax]
	[-nocpp] [-G num] [-mcpu=CPU] (if implemented)
	[--trap] [--break]
	[--emulation=name]
	[-- | files...]

These assembler switch options are discussed in depth within the manual, *MC-Utilities.pdf* (docs directory on i.MX GNU X-Tools CD-ROM and in Cygwin/docs). In addition, each assembler has target specific switch options. The target-specific switch options are located in the Appendices of this User Guide.

Unlike older assemblers, as is designed to assemble a source program in one pass of the source file. This has a subtle impact on the '.org' directive. These options enable listing output from the assembler. By itself, `-a' requests high-level, assembly, and symbols listing. You can use other letters to select specific options for the list: `-ah' requests a high-level language listing, `-al' requests an output-program assembly listing, and `-as' requests a symbol table listing. High-level listings require that a compiler debugging option like `-g' be used, and that assembly listings (`-al') be requested also.

Use the `-ac' option to omit false conditionals from a listing. Any lines which are not assembled because of a false '.if' (or '.ifdef', or any other conditional), or a true '.if' followed by an '.else', will be omitted from the listing. Use the `-ad' option to omit debugging directives from the listing.

Once you have specified one of these options, you can further control listing output and its appearance using the directives ‘.list’, ‘.nolist’, ‘.psize’, ‘.eject’, ‘.title’, and ‘.sbttl’. The `-an’ option turns off all forms processing. If you do not request listing output with one of the `-a’ options, the listing-control directives have no effect. The letters after `-a’ may be combined into one option, e.g., `-aln’.

-a[cdhlmns]
Turn on listings, in any of a variety of ways:

-ac	Omit false conditionals
-ad	Omit debugging directives
-ah	Include high-level source
-al	Include assembly
-am	Include macro expansions
-an	Omit forms processing
-as	Include symbols
=file	Set the name of the listing file

You may combine these options; for example, use ‘-aln‘ for assembly listing without forms processing. The ‘=file‘ option, if used, must be the last one. By itself, ‘-a’ defaults to ‘-ahls‘.

-D	Ignored*
--defsym sym= value	Define the symbol sym to be value before assembling the input file. value must be an integer constant. As in C, a leading 0x indicates a hexadecimal value, and a leading 0 indicates an octal value.
-f	“Fast”—skip white space and comment preprocessing (assume source is compiler output). Warning: if you use `-f’ when the files actually need to be preprocessed (if they contain comments, for example), as does not work correctly.
--gstabs	Generate stabs debugging information for each assembler line. This may help debugging assembler code, if the debugger can handle it.
--help	Print a summary of the command line options and exit.
-I *dir*	Add directory, ‘dir’, to the search list for ‘.include’ directives. You may use -I as many times as necessary to include a variety of paths. The current working directory is always searched first; after that, as searches any `-I’ directories in the same order as they were specified (left to right) on the command line.
-J	Don’t warn about signed overflow
-K	Issue warnings when difference tables altered for long displacements.
-L	Keep (in the symbol table) local symbols, starting with L.
--keep-locals	Keep (in the symbol table) local symbols. On traditional ‘a.out’ systems these start with L, but different systems have different local label prefixes.
-M or --mri	This changes the syntax and pseudo-op handling of as to make it compatible with the ASM68K or the ASM960 (depending upon the configured target) assembler from Microtec Research.
--MD	as can generate a dependency file for the file it creates. This file consists of a single rule suitable for make describing the dependencies of the main source file.
-o	objfile Name the object-file output from as objfile. By default it has the name `a.out’ (or `b.out’, for Intel 960 targets only). You use this option (which takes exactly one filename) to give the object file a different name.
-R	Fold the data section into the text section.
--statistics	Print the maximum space (in bytes) and total time (in seconds) used by assembly.

--strip-local-absolute	Remove local absolute symbols from the outgoing symbol table.
-v -version	Print the as version.
--version	Print the as version and exit.
-W	Suppress warning messages.
-w	Ignored*
-x	Ignored*
-z	Generate an object file even after errors.
-- | files...	Standard input, or source files (files ...) to assemble.

* This option is accepted for script compatibility with calls to other assemblers.

4.9 Using the Linker

The ld linker combines several object modules and libraries into a single executable file. It resolves references to external variables, external procedures, and libraries, creating a complete, self-sufficient program. You never need to invoke ld explicitly. In most cases, it is simpler to use the GCC command to link files, even if you do not have any source files to compile. The GCC guarantees that certain libraries will be present in the proper order even if they are not listed on the command line. If you use ld as a linker, you need to mention these libraries explicitly.

4.9.1 Invoking ld

The rules for invoking ld, if you must do so, are the same as for GCC or as. The basic ld command is as follows:

<target-alias>$ ld <list-of-options> <list-of-files-and–libraries> (enter)

Where 'list-of-files-and-libraries' is a series of filenames and library specifications. To include a library in this list, use the notation '–lname', where the name of the library file is either '/lib/libname.a' or '/usr/lib/libname.a'. The linker processes the 'list-of-files-and-libraries' in order. When it reaches a library, it extracts only those modules that it currently needs to resolve external references. Consequently, the position in which libraries appear in this list is important. For example, the command:

<target-alias>$ ld prog1.o –lm prog2.o (enter)

results in an 'Undefined symbol' message if 'prog2.o' refers to any programs in the library '/usr/lib/libm.a' – unless you happen to be lucky and 'prog2.o' only uses routines that the linker extracted for the sake of 'prog1.o'. Note that libraries may refer to other libraries; thus, the command:

<target-alias>$ ld prog1.o –lat –lfo (enter)

leads to 'Undefined symbol' messages if the 'fo' library requires any routines from 'at'.

This situation is more complex for a user-generated library. Such a library should contain an index, so that the linker can find each module regardless of its order within the library. Some systems always generate an index when you create or modify the library with the ar command (the GNU ar does this). On other systems you have to put in the index yourself by using the ranlib command.

If you want to create an executable file, the beginning of the first file in the list-of-files must be the program's entry point. This is not the same as the apparent entry point to your C source program. Before your program begins executing, the computer must execute a standard run-time initialization routine. To ensure that this is in place, '/lib/crt0.o' must be the first file in the 'list-of-files-and-libraries'. This ensures that this initialization routine is linked to your program.

Alternatively, you can link by using the GCC command without any C source files. When GCC invokes the linker, it automatically adds ‘crt0.o’ and many other libraries in the proper place. For example, the command ‘gcc exp.o’ generates the following ld command:

<target-alias>$ ld –dc –dp –e start –X o –o a.out /usr/<target-alias>/lib/crt0.o –lc (enter)

In this command, the run-time initialization module ‘/usr/<target-alias>/lib/crt0.o’ appears explicitly, in addition to requests to resolve references to the C library (the general runtime library). You can see what ld command is generating when you compile a program on your system, by invoking GCC with the ‘–v’ (verbose) option.

4.9.2 Linker Options

The GCC passes any options it does not recognize to the linker. The most important options can therefore be placed directly on the GCC command line. These options are:

-o name.x

Instead of naming the executable output file ‘a.out’ or ‘a.exe’ (native builds under Cygwin), it names it ‘name.x’. Other popular naming extensions for embedded applications include: ‘name.elf’, ‘name.coff’, ‘name.srec’, and ‘name.bin’. Microcross has standardized all of its cross-executable names to the ‘.x’ extension.

-lname

Link the program to the library named libname.a. The linker looks in the directories ‘/lib’ and ‘/usr/lib’ to find this library. Note that the GNU linker truncates the name’s prefix and extension (i.e., ‘lib’ and ‘.a’ are not necessary for ‘libname.a’ as a linked library file. If you create your own libraries to link into your program, you will need to name them ‘lib**name**.a’ and link in as ‘-l**name**’.

-Ldir

To find any libraries, look in the directory dir before looking in the standard library directories ‘/lib’ and ‘/usr/lib’.

-s

Remove the symbol table from the executable output file. This makes the output file significantly smaller, but makes debugging nearly impossible. Therefore, this option should not be used until the program works successfully. Note that using the program strip has the same effect.

-x

Remove all local symbols from the output file. Global symbols (subprograms and common block names) remain in the output file. This reduces the object file’s size. Ignored unless ‘–s’ is specified.

-n
Make the text segment read-only.

-r
Create an object file that can be included in further linking runs (i.e., further passes through ld). Among other things, this inhibits 'Undefined symbol' messages (because the symbols may be defined in a later ld pass) and prevents ld from creating common storage immediately. If you wish to create common storage at this time, use the '–d' option also.

-e name
Use the symbol name as the entry point to the executable program. By default, the entry point is the beginning of the first object module. The GCC automatically links your object files with a run-time initialization module ('/usr/lib/crt0.o') that starts your program and provides its initial entry point. If you run the linker separately, you must either put '/usr/lib/crt0.o' at the start of your object files, or provide your own entry point.

-M
Produce a load map that shows where each function is located in the resulting object file, where each section begins and ends, and what the value of each global symbol is. This option is usually used with GCC as follows: '-Wl,-M=<mapfile-name>', where '<mapfile-name>' can be any name you specify.

-b format
Read object modules in the given format. To get a list of formats that ld understands, give the command 'objdump –l' and near the bottom of the options list are the supported targets and architectures. This can be helpful in some cross-development situations. The '–b' option applies to all object files and libraries following it on the command line, until another '–b' option appears. In theory, you can use this feature to link objects from several different formats into a single executable.

-oformat format
Create object modules in the given format. Again, 'objdump –l' gives you a list of formats that ld understands. The ld is configured to produce the most reasonable output format for its target machine. Its assumptions about what is "reasonable" are probably true about 99.99 percent of the time. But there may be special-purpose situations in which you would want another output format.

Here is an example of a customized ld command:

<target-alias>$ ld –r –o bigofile.o prog1.o prog2.o –lmylib (enter)

This command links the files 'prog1.o' and 'prog2.o' and the library file '/usr/lib/libmylib.a'. The resulting file is named 'bigofile.o'; it can be linked further and may still contain unresolved references.

4.9.3 Linker Scripts

One advanced feature of the GNU linker is its ability to work from scripts written in its own command language. If you are a true masochist, you might be able to avoid running GCC altogether; you might be able to implement your own compiler as a linker script. There are a few situations in which you would actually need a linker script, but you should be aware that they exist for purposes like:

- Gaining tight control over the format of the output file – perhaps so an embedded application will fit into the smallest possible ROM, perhaps to optimize link order.
- Supporting an object format that ld does not provide – perhaps an object format of your own design, or an object format for some special-purpose operating system.

Microcross recommends starting with the default linker script and modifying it to meet your needs. To extract the standard linker script into a file, type the command:

<target-alias>$ ld –verbose >linker.ld (enter)

This command will create a file named 'linker.ld', and you will need to open this file in an editor to delete the header lines before and including'======' and the footer single line '======'. Then the script file can be saved or copied into a source directory and be used with the GCC option '-Wl,-T,linker.ld' to replace the standard linker script. Now, you can modify the linker script with confidence that you have a working startup script.

4.9.4 Link-Order Optimization

If you have done a lot of development work, you have probably noticed that the order in which you link your files can have a significant effect on performance. By changing the link order, you are changing the way the executable file lies in the instruction cache. The cache is a fast area of memory that stores pages of instructions so that the processor does not have to go back to slower parts of memory (or even worse, the disk) for every new instruction. Certain link orders minimize instruction-cache miss. The effect usually is not large, but in pathological cases (a really bad link order on a machine that is very sensitive to cache miss), link optimization can speed up runtime by 50 percent.

Unfortunately, not much can be said about link-order optimization. There are few rules, if any; and all the rules have many exceptions. In general, it is a good idea to place modules that call each other near each other on the command line. The reasoning behind the heuristic is simple: if function A makes many calls to function B, and both A and B can fit into the cache simultaneously, you will not pay a penalty for cache miss. Your best chance of fitting both functions into the cache simultaneously occurs when they are located next to each other in the object file. The '–M' option, which produces a load map, shows you how the object file is arranged; it will help you investigate cache performance.

Normally, rearranging the order of the object modules on the command line is sufficient for experimenting with link-order optimization. However, you can get very fine control over your executable file by writing a linker script. If you have a thorough knowledge of your target machine's architecture, you may be able to use this to your advantage – though you will probably reach the point of diminishing returns fairly quickly.

4.9.5 The C Runtime (crt0)

To link and run C or C++ programs, you need to define a small module, usually written in assembly as 'crt0.s', but sometimes written as a C file as 'crt0.c', to initialize hardware using C conventions before calling main. There are some examples available in the sources of Newlib C library; perform a simple search through the source tree for 'crt0.s' code, as well as examples of system calls with sub-routines.

To write your own 'crt0.s' or 'crt0.c' module, you need specific information about your target – see 'MC-Embedded-Systems.pdf' in the i.MX GNU X-Tools CD docs directory (also in Cygwin/docs) for details on how to create the C runtime environment.

4.10 Object Translation (ELF to Binary, SREC, etc.)

Most of the i.MX GNU X-Tools toolsuites produce an Extended Linker Format (ELF) object file as the default format, and it can be used during debug and testing; however, the final code needs to be stripped and translated to another format. It is beyond the scope of the user guide to explain all of the various formats, which include COFF, ECOFF, binary, srec, tekhex, ihex, symbolsrec, etc..

Here is a quick example of how to use objcopy; refer to the binutils documentation in the docs directory on the distribution CD or in the Cygwin/docs directory for details on how to use. From the Shell, issue the following commands (substitute your target alias and bfdname as necessary). To see what bfdnames are available, issue the command '<target-alias>-objcopy' (enter). All of the objcopy options appear in the Shell, and at the bottom of the screen is a list of the target bfdnames that can be used to specify the input and output formats. The ARM, for example, has 9 bfdname names: elf32-littlearm, elf32-bigarm, elf32-little, elf32-bit, srec, symbolsrec, tekhex, binary, and ihex. Depending on the compiler options used, the input bfdname will change (e.g., compile for big endian as opposed to little endian). The default output of the ARM is elf32-littlearm or elf32-little.

$ xtools arm-elf (enter)
arm-elf$ cd /home/test (enter)
arm-elf$ gcc –o div.x div.c (enter)
arm-elf$ objcopy –S –I elf32-littlearm –O binary div.x div.bin (enter)

Note: The '-S' strips all symbols out to make the object as small as possible. The '-I' is specifying the input target format (bfdname), and the '-O' specifies the output target format (bfdname). The input file is 'div.x' and the output file is 'div.bin'.

4.11 Creating/Updating Libraries

The command ar creates libraries (or archives) of object modules. They are similar to the Unix / Linux utilities with the same names, except that you do not need a separate ranlib. This section gives a brief description of how to use these utilities in the i.MX GNU X-Tools command shell.

In naming a library for use with i.MX GNU X-Tools, always use the prefix 'lib' and suffix extension 'a' with your static library name; this is the GNU convention and is necessary for GCC to automatically scan the library at link time, so the syntax for linking a particular library, say libm.a the math library, is -lm.

To create a new library, use the ar command, as follows:

<target-alias>$ ar -rs lib<name>.a list-of-files (enter)

The option 'r' indicates that the command ar should add the files named in the 'list-of-files' to the library named 'name', creating a new library if necessary. If a file is mentioned twice in the 'list-of-files', ar includes it in the archive twice. The 's' option tells ar to produce an index for the archive; this is the function that ranlib would perform. If you include the 's' option whenever you create or modify a library, you will not need to use ranlib.

To update a library, use the command:

<target-alias>$ ar -rus lib<name>.a list-of-files (enter)

This compares the dates of any listed files with the version of the file in the library. If the file in 'list-of-files' is more recent than the version contained in the library, ar substitutes the newer version for the older version. The 's' option updates the library's index.

To delete one or more files from a library, use the command:

<target-alias>$ ar -ds lib<name>.a list-of-files (enter)

This option deletes all the files found in 'list-of-files'.

To extract one or more files from a library, use the command:

<target-alias>$ ar -x lib<name>.a list-of-files (enter)

This does not modify the library file itself. It extracts the files named in the 'list-of-files' from the library, regenerating them in the current directory with their original names. Normally, the timestamp of the extracted files is the time at which ar recreated them. If you use the option 'xo' instead of 'x', ar sets the timestamp of the extracted files to the time recorded in the archive.

You can still create an ordered (index-less) library with ar and invoke ranlib as a separate step if you want, which is the convention of older Unix Systems. However, there is no longer any good reason for doing that when using GNU tools.

4.12 GNU Libraries

If you are familiar with Unix / Linux and C programming, the libraries you will find in the i.MX GNU X-Tools development environment should not confuse you. The libraries you expect will all be there: standard I/O, the math library, the strings library, etc.. The libraries are ANSI C and POSIX compliant. Moreover, there are many functions that Unix / Linux programmers expect, but are not specified by either of these standards.

Some C functions have been standardized and found in the GNU environment, but system calls and math libraries are not the same as those found in a typical DOS / Windows environment. You will have to re-learn these functions. A good resource for learning the GNU Libraries is on the i.MX GNU X-Tools CD 'Docs' directory: 'MC-Libraries.pdf' and 'Math_lib.pdf'.

4.13 Instruction Set Simulator (ISS) Options

Refer to Table 4.6 for your target's required ISS CFLAGS and LFLAGS. The CFLAGS and LFLAGS my be necessary for compiling an application to run in the simulator. When using GCC to control the link step, use the option, '-Wl,-<linker-flag>', to pass the option to the linker.

Table 4.6 Required CFLAGS / LFLAGS for i.MX GNU X-Tools Simulator Builds

Target Alias	Tool Name	Compiler Flags	Linker Flag shown with –Wl,-<linker-flag> for passing options using GCC
arm-elf	arm-elf-gcc/g++	None	None

Section 5. How to Use Command Line GDB

5.1 Summary of GDB, the GNU Debugger

The purpose of a debugger such as the GNU debugger, GDB, is to allow you to see what is going on inside another program while it executes—or what another program was doing at the moment it stopped. The GDB can do four things to help you catch “bugs.”

- Start your program, specifying anything that might affect its behavior.
- Make your program stop on specified conditions.
- Examine what has happened when your program has stopped.
- Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another problem affecting your program.

5.1.1 GDB as Free Software

The GNU debugger, GDB, is *free softwar*e, protected by the GNU General Public License (GPL). The GPL gives you the freedom to copy or adapt a licensed program—but every person getting a copy also gets with it the freedom to modify that copy (which means that they must get access to the source code), and the freedom to distribute further copies. Typical software companies use copyrights to limit your freedoms; the Free Software Foundation uses the GPL to preserve these freedoms. Fundamentally, the General Public License is a license that says you have these freedoms and that you cannot take these freedoms away from anyone else. To see the GNU General Public License, see ‘license.txt’ on i.MX GNU X-Tools CD-ROM.

5.1.2 Requirements of GDB

Before using GDB, you should understand the formal requirements and other expectations for GDB. Although some of these may seem obvious, there have been proposals for GDB that have run counter to these requirements. First of all, GDB is a debugger. It’s not designed to be a front panel for embedded systems. It’s not a text editor. It’s not a shell. It’s not a programming environment. GDB is an interactive tool. Although a batch mode is available, GDB’s primary role is to interact with a human programmer. The GDB should be responsive to the user. A programmer hot on the trail of a nasty bug, and operating under a looming deadline, is going to be very impatient of everything, including the response time to debugger commands. GDB should be relatively permissive, such as for expressions. While the compiler should be picky (or have the option to be made picky), since source code lives for a long time usually, the programmer doing debugging shouldn’t be spending time figuring out to mollify the debugger. GDB will be called upon to deal with really large programs. Executable sizes of 50 to 100 megabytes occur regularly, and there are reports of programs approaching 1 gigabyte in size. GDB should be able to run everywhere. No other debugger is available for even half as many configurations as GDB supports.

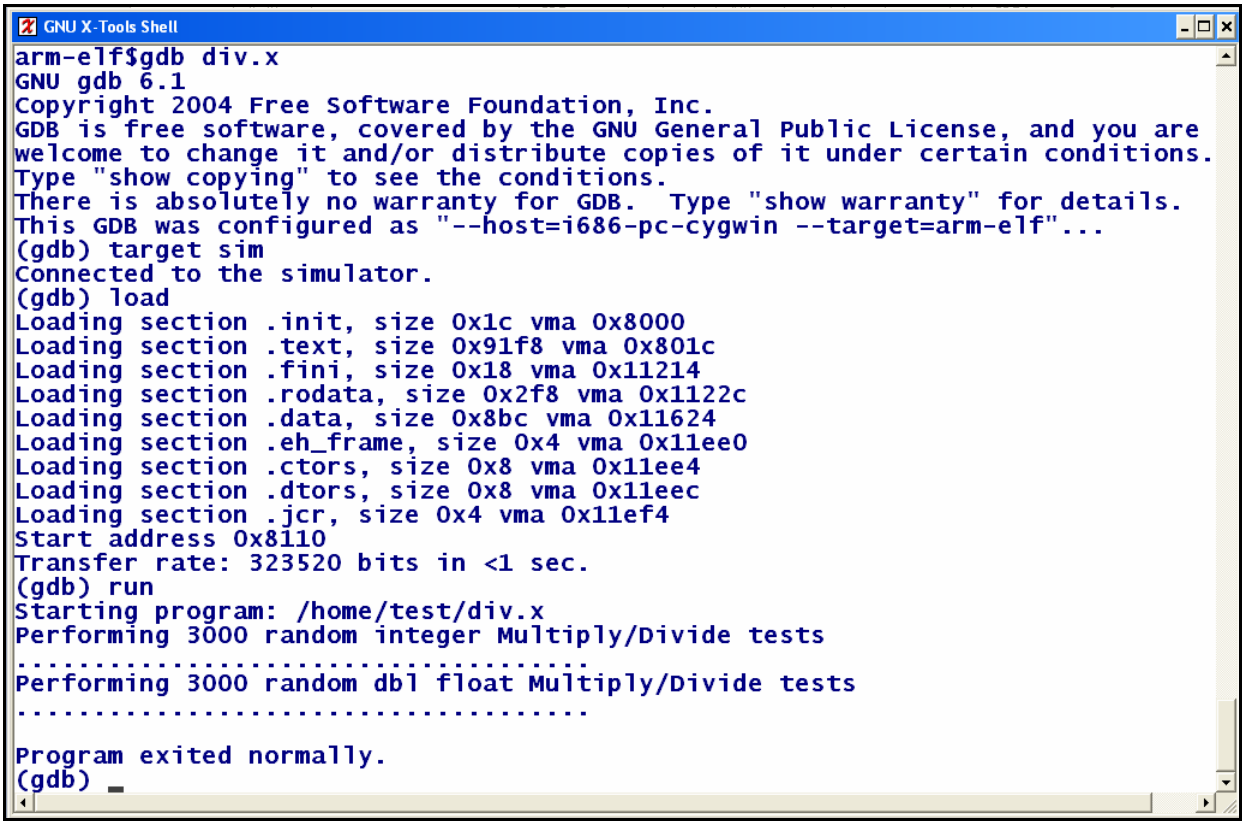
5.1.3 Startup GDB

To start a GDB debugging session, open a i.MX GNU X-Tools bash shell (or xterm on Linux) and issue the following commands:

$ xtools <target-alias> (enter)
<target-alias>$ gdb <program-name> (enter)

Its that simple to start GDB and load a program for any one of the toolsuites supported by Microcross. See Figure 5.1, which shows GDB for arm-elf opening a binary and running a program called 'div.x' through the simulator. We recommend getting acquainted with 'mc-debug.pdf' for details on how to use GDB, if you are not familiar with some of the commands presented in the Reference Card below. The GDB as well as C Reference Cards can be downloaded from the following web site: http://www.refcards.com.

Figure 5.1 A GDB Example in View


GNU X-Tools Shell
arm-elf$gdb div.x
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "--host=i686-pc-cygwin --target=arm-elf"...
(gdb) target sim
Connected to the simulator.
(gdb) load
Loading section .init, size 0x1c vma 0x8000
Loading section .text, size 0x91f8 vma 0x801c
Loading section .fini, size 0x18 vma 0x11214
Loading section .rodata, size 0x2f8 vma 0x1122c
Loading section .data, size 0x8bc vma 0x11624
Loading section .eh_frame, size 0x4 vma 0x11ee0
Loading section .ctors, size 0x8 vma 0x11ee4
Loading section .dtors, size 0x8 vma 0x11eec
Loading section .jcr, size 0x4 vma 0x11ef4
Start address 0x8110
Transfer rate: 323520 bits in <1 sec.
(gdb) run
Starting program: /home/test/div.x
Performing 3000 random integer Multiply/Divide tests
.....
Performing 3000 random dbl float Multiply/Divide tests
.....
Program exited normally.
(gdb)

5.1.4 Startup of GDB with DDD

To use the Data Display Debugger (DDD) with the i.MX GNU X-Tools command line version of GDB, follow these easy steps to get started, and refer to distribution documentation for details -- 'ddd.pdf' in the docs directory on the distribution CD or in Cygwin/docs.

Start 'XWin' using the desktop icon (in Windows), which was created by the i.MX GNU X-Tools installation manager. In Linux, refer to your distribution's documentation for installing and invoking DDD. We are assuming a Windows host for the following procedures.

Issue the following commands in the XWindows Shell (substitute your target alias as appropriate):

Example

$ xtools arm-elf (enter)
arm-elf$ cd /home/test (enter)
arm-elf$ gcc –g –o div.x div.c (enter)
arm-elf$ ddd div.x –debugger arm-elf-gdb (enter)

Start DDD In General

$ ddd <name-of-elf-object> -debugger <target-alias>-gdb (enter)

This concludes our short DDD example. There is a complete user guide on DDD in the docs directory on the i.MX GNU X-Tools CD-ROM.

5.2 GDB Comprehensive Quick Reference

Table 5.1. Essential Commands

Command Syntax	Description
GDB *program* [*core]*	debug *program* [using coredump *core*]
b [file:]function	set breakpoint at *function* [in *file]*
run [*arglist*]	start your program [with *arglist*]
bt	backtrace: display program stack
p *expr*	display the value of an expression
c	continue running your program
n	next line, stepping over function calls
S	next line, stepping into function calls
info stack	view the call stack
print *(int*)0x10000	display memory location
set *(int*)0x1000=0x123	set memory
info registers	display all CPU registers
print $<reg-name>	displays register value
set $<reg-name>=<value>	sets register value
step	runs next line of code (step into)
stepi	runs next instruction
next	runs next instruction, but doesn't enter (step over)
info br	list breakpoints
br Init	insert breakpoint on the Init function
br 33	insert breakpoint on line # 33
list Init	show source code listing
disassemble	shows assembly code of higher level code
file <name-of-object>	loads file for debugging
load	loads object sections
target <name>	selects <name> = remote, serial, sim, gdbserver, etc.

run	run a program in the simulator -- before using this command, a sequence must be executed as follows:
	$ <target-alias>-gdb <filename> (object file with debug symbols)
	$ target sim
	$ load
	$ run

x / FMT	display modifier – where FMT is x – print in hex d – print in decimal s – print as string w – print in 32-bit words h – print in 16-bit words b – print in 8-bit words # - print number of items

Table 5.2. Starting GDB

Command Syntax	Description
GDB	start GDB, with no debugging files
GDB *program*	begin debugging *program*
GDB *program core*	debug coredump *core* produced by *program*
GDB --help	describe command line options

Table 5.3. Stopping GDB

Command Syntax	Description
quit	exit GDB; also q or EOF (e.g., C-d)
INTERRUPT	(eg C-c) terminate current command, or send to running process

Table 5.4. Getting Help

Command Syntax	Description
help	list classes of commands
help *class*	one-line descriptions for commands in *class*
help *command*	describe *command*

Table 5.5. Executing your Program

Command Syntax	Description
run *arglist*	start your program with *arglist*
run	start your program with current argument list
run…*<inf >outf*	start your program with input, output redirected
kill	kill running program
tty *dev*	use *dev* as stdin and stdout for next run
set args *arglist*	specify *arglist* for next run
set args	specify empty argument list
show args	display argument list
show env	show all environment variables
show env *var*	show value of environment variable var
set env *var string*	set environment variable *var*
unset env *var*	remove *var* from environment

Table 5.6. Shell Commands

Command Syntax	Description
cd dir	change working directory to dir
pwd	print working directory
make …	call \make"
shell *cmd*	execute arbitrary shell command string

Table 5.7. Breakpoints and Watchpoints

Command Syntax	Description
break [*file*:]line	set breakpoint at line number [in *file]*
B [*file*:]line	e.g.: break main.c:37
break [*file*:]*func*	set breakpoint at *func* [in *file*]
break +*offset*	set break at *offset* lines from current stop
break -*offset*	set break at *offset* lines from current stop
break *addr	set breakpoint at address *addr*
break	set breakpoint at next instruction
break …if *expr*	break conditionally on nonzero *expr*

Command Syntax	Description
cond n [*expr*]	new conditional expression on breakpoint *n*; make unconditional if no *expr*
tbreak…	temporary break; disable when reached
rbreak *regex*	break on all functions matching *regex*
watch *expr*	set a watchpoint for expression *expr*
catch *event*	break at *event*, which may be catch, throw, exec, fork, vfork, load, or unload
info break	show defined breakpoints
info watch	show defined watchpoints
Clear	delete breakpoints at next instruction
clear [*file*:]*fun*	delete breakpoints at entry to *fun()*
clear [*file*:]*line*	delete breakpoints on source line
delete [*n*]	delete breakpoints [or breakpoint *n*]
disable [*n*]	disable breakpoints [or breakpoint *n*]
enable [*n*]	enable breakpoints [or breakpoint *n*]
enable once [*n*]	enable breakpoints [or breakpoint *n*]; disable again when reached
enable del [*n*]	enable breakpoints [or breakpoint *n*]; delete when reached
ignore *n count*	ignore breakpoint *n, count* times
commands *n* [silent] command-list	execute GDB *command-list* every time breakpoint *n* is reached. [silent suppresses default display]
End	end of *command-list*

Table 5.8. Program Stack

Command Syntax	Description
backtrace *[n]*	print trace of all frames in stack; or of *n*
bt *[n]*	frames|innermost if *n*>0, outermost if *n*<0
frame *[n]*	select frame number n or frame at address *n*; if no *n*, display current frame
up *n*	select frame *n* frames up
down *n*	select frame *n* frames down
info frame [*addr*]	describe selected frame, or frame at *addr*
info args	arguments of selected frame
info locals	local variables of selected frame
info reg [*rn*]…	register values [for regs *rn*] in selected
info all-reg [*rn*]	frame; all-reg includes floating point

Table 5.9. Execution Control

Command Syntax	Description
continue [*count*] c [*count*]	continue running; if count specified, ignore this breakpoint next count times
step [*count*] s [*count*]	execute until another line reached; repeat count times if specified
stepi [*count*] si [*count*]	step by machine instructions rather than source lines
next [*count*] n [*count*]	execute next line, including any function calls
nexti [*count*] ni [*count*]	next machine instruction rather than source line
until [*location*]	run until next instruction (or location)
finish	run until selected stack frame returns

Command Syntax	Description
return [*expr*]	pop selected stack frame without executing [setting return value]
signal *num*	resume execution with signal s (none if 0)
jump *line* jump **address*	resume execution at specified line number or address
set var=*expr*	evaluate expr without displaying it; use for altering program variables

Table 5.10. Display

Command Syntax	Description
print [*/f*] [*expr*] p [/f] [*expr*]	show value of expr [or last value $] according to format *f:*
x	Hexadecimal
d	signed decimal
u	unsigned decimal
o	octal
t	binary
a	address, absolute and relative
c	character
f	floating point
call [*/f*] *expr*	like print but does not display void
x [*/Nuf*] *expr*	examine memory at address expr; optional format spec follows slash
N	count of how many units to display
u	unit size; one of b individual bytes h halfwords (two bytes) w words (four bytes) g giant words (eight bytes)
f	printing format. Any print format, or s null-terminated string i machine instructions
disassem [*addr*]	display memory as machine instructions

Table 5.11. Automatic Display

Command Syntax	Description
display [/f] expr	show value of expr each time program stops [according to format *f*]
display	display all enabled expressions on list
undisplay *n*	remove number(s) *n* from list of automatically displayed expressions
disable disp *n*	disable display for expression(s) number *n*
enable disp *n*	enable display for expression(s) number *n*
info display	numbered list of display expressions

Table 5.12. Expressions

Command Syntax	Description
expr	an expression in C, C++, or Modula-2 (including function calls)
addr@*len*	an array of *len* elements beginning at *addr*
file::*nm*	a variable or function nm defined in *file*
{*type*}*addr*	read memory at *addr* as specified *type*
$	most recent displayed value
$*n*	*n*th displayed value

Command Syntax	Description
$$	displayed value previous to $
$$*n*	nth displayed value back from $
$_	last address examined with x
$__	value at address $_
$*var*	convenience variable; assign any value
show values [*n*]	show last 10 values [or surrounding $*n*]
show conv	display all convenience variables

Table 5.13. Symbol Table

Command Syntax	Description
info address *s*	show where symbol *s* is stored
info func [*regex*]	show names, types of defined functions (all, or matching *regex*)
info var [*regex*]	show names, types of global variables (all, or matching *regex*)
whatis [*expr*] ptype [*expr]*	show data type of *expr* [or $] without evaluating; ptype gives more detail
ptype *type*	describe *type*, struct, union, or enum

Table 5.14. GDB Scripts

Command Syntax	Description
source *script*	read, execute GDB commands from file *script*
define cmd *command-list*	create new GDB command cmd; execute script defined by command-list
end	end of command-list
document *cmd* *help-text*	create online documentation for new GDB command *cmd*
end	end of *help-text*

Table 5.15. Signals

Command Syntax	Description
handle signal act	specify GDB actions for signal:
print	announce signal
noprint	be silent for signal
stop	halt execution on signal
nostop	do not halt execution
pass	allow your program to handle signal
nopass	do not allow your program to see signal
info signals	show table of signals, GDB action for each

Table 5.16. Debugging Targets

Command Syntax	Description
target *type* param	connect to target machine, process, or file
help target	display available targets
attach param	connect to another process
detach	release target from GDB control

Table 5.17. Controlling GDB

Command Syntax	Description
set param value	set one of GDB's internal parameters
show param	display current setting of parameter

Parameters understood by set and show:

Command Syntax	Description
complaint *limit*	number of messages on unusual symbols
confirm *on/off*	enable or disable cautionary queries
editing *on/off*	control readline command-line editing
height *lpp*	number of lines before pause in display
language *lang*	Language for GDB expressions (auto, c or modula-2)
listsize *n*	number of lines shown by list
prompt *str*	use *str* as GDB prompt
radix *base*	octal, decimal, or hex number representation
verbose *on/off*	control messages when loading symbols
width *cpl*	number of characters before line folded
write *on/off*	Allow or forbid patching binary, core files (when reopened with exec or core)
history… h… h exp *off/on* h file *filename* h size *size* h save *off/on*	groups with the following options: disable/enable readline history expansion file for recording GDB command history number of commands kept in history list control use of external file for command history
print… p… p address *on/off* p array *off/on* p demangl *on/off* p asm-dem *on/off* p elements *limit* p object *on/off* p pretty *off/on* p union *on/off* p vtbl off/on	groups with the following options: print memory addresses in stacks, values compact or attractive format for arrays source (demangled) or internal form for C++ symbols demangle C++ symbols in machine-instruction output number of array elements to display print C++ derived types for objects struct display: compact or indented display of union members display of C++ virtual function tables
show commands	show last 10 commands
show commands *n*	show 10 commands around number *n*
show commands +	show next 10 commands

Table 5.18. Working Files

Command Syntax	Description
file [*file*]	use file for both symbols and executable; with no arg, discard both
core [*file*]	read file as coredump; or discard
exec [*file*]	use file as executable only; or discard
symbol [*file*]	use symbol table from file; or discard
load *file*	dynamically link file and add its symbols
add-sym *file addr*	read additional symbols from *file*, dynamically loaded at *addr*
info files	display working files and targets in use
path *dirs*	add *dirs* to front of path searched for executable and symbol files
show path	display executable and symbol file path
info share	list names of shared libraries currently loaded

Table 5.19. Source Files

Command Syntax	Description
dir *names*	add directory *names* to front of source path
dir	clear source path
show dir	show current source path
list	show next ten lines of source
list -	show previous ten lines
list lines [*file*:]*num* [*file*:]*function* +*off* -*off* **address*	display source surrounding lines, specified as: line number [in named file] beginning of function [in named file] *off* lines after last printed *off* lines previous to last printed line containing *address*
list *f,l*	from line *f* to line *l*
info line *num*	show starting, ending addresses of compiled code for source line num
info source	show name of current source file
info sources	list all source files in use
forw *regex*	search following source lines for *regex*
rev *regex*	search preceding source lines for *regex*

Table 5.20. GDB under GNU Emacs

Command Syntax	Description
M-x GDB	run GDB under Emacs
C-h m	describe GDB mode
M-s	step one line (step)
M-n	next line (next)
M-i	step one instruction (stepi)
C-c C-f	finish current stack frame (finish)
M-c	continue (cont)
M-u	up arg frames (up)
M-d	down arg frames (down)
C-x &	copy number from point, insert at end
C-x SPC	(in source file) set break at point

Section 6. How to Use Visual GDB Debugger

6.1 Using Visual GDB Debugger

Microcross builds and packages the popular GDBTK (a.k.a. Insight™) and calls it Visual GDB™ to differentiate the product from the FSF (GDBTK) and other vendors' products (Red Hat Insight®). Visual GDB has all of the features accustomed to Red Hat Insight, and we provide a short introduction on how to use the features of Visual GDB.

6.2 Visual GDB, An Alternative Interface to Command Line

The following documentation serves as a general reference for i.MX GNU X-Tools' graphical user interface, its visual debugger, Visual GDB; for more information, see also Visual GDB's Help menu for discussion of general functionality and use of menus, buttons or other features and Examples of Debugging with Visual GDB in this Section.

If using a i.MX GNU X-Tools Bash Shell, compile your program into a cross-executable. In Figure 5.1, we started the i.MX GNU X-Tools Bash Shell and show a simple demonstration using the arm-elf toolsuite to create a cross-executable for loading into Visual GDB.

Note: GDBTK is the Visual GDB executable at the command line. Also shown in Figure 6.1 is the filename of the cross-executable, 'pascal.x'. A simple 'GDB.ini' file can be configured in the source and executable's file directory. The 'GDB.ini' file can include the cross-executable's filename, board register initialization settings, and any other initialization settings necessary for startup that may be needed to perform on-chip debug (e.g., debug agents used with i.MX GNU X-Tools include the Abatron BDI2000 and Macraigor Systems' Wiggler/Raven/mpDemon) – see documentation listed in the references that describe using GDB.

Figure 6.1. Program to debug window

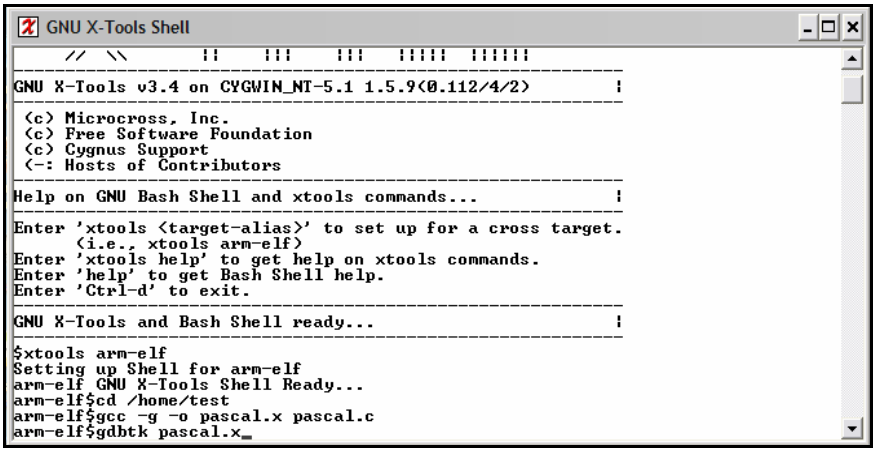

GNU X-Tools Shell
GNU X-Tools v3.4 on CYGWIN_NT-5.1 1.5.9(0.112/4/2)
(c) Microcross, Inc.
(c) Free Software Foundation
(c) Cygnus Support
(-: Hosts of Contributors
Help on GNU Bash Shell and xtools commands...
Enter 'xtools <target-alias>' to set up for a cross target.
(i.e., xtools arm-elf)
Enter 'xtools help' to get help on xtools commands.
Enter 'help' to get Bash Shell help.
Enter 'Ctrl-d' to exit.
GNU X-Tools and Bash Shell ready...
$xtools arm-elf
Setting up Shell for arm-elf
arm-elf GNU X-Tools Shell Ready...
arm-elf$cd /home/test
arm-elf$gcc -g -o pascal.x pascal.c
arm-elf$gdbtk pascal.x

Figure 6.1 shows the following commands using an ARM toolsuite (substitute your target alias if different):

Start i.MX GNU X-Tools Shell (Windows) or 'xterm' (Linux)
Issue these commands:

$xtools arm-elf (enter)
arm-elf$ cd /home/test (or appropriate directory) (enter)
arm-elf$ gcc –g –o pascal.x pascal.c (enter)
arm-elf$ gdbtk pascal.x (enter)

The Visual GDB debugger opens the ELF object file that was compiled with the '-g' debug option. To run the program using the simulator, click on the Run icon and when the Target dialog box opens, select Simulator as the Target. To familiarize yourself with all of the features of Visual GDB, read on.

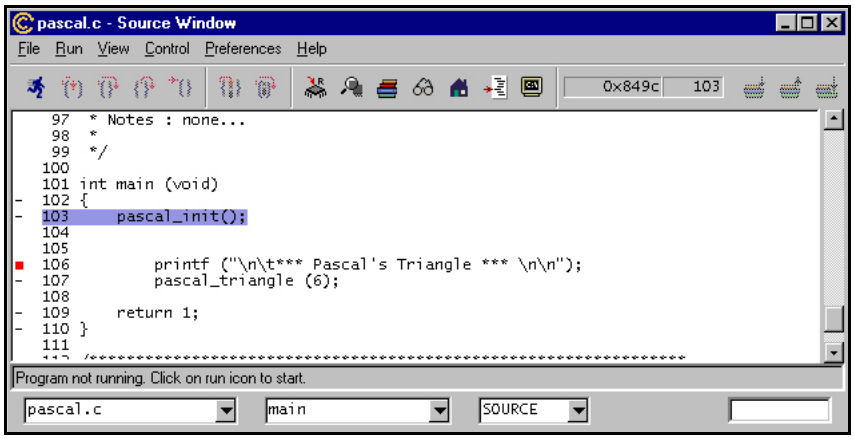
Note: Examples of using JTAG hardware debuggers with Visual GDB are at the end of this section. We have selected two popular JTAG debuggers to show examples: Abatron BDI2000 and Macraigor mpDemon.

WARNING: Having an inactive debugging session open when starting another debugging session with i.MX GNU X-Tools will close all projects. All work will be unrecoverable.

Using the Source Window

When Visual GDB first launches, it displays an empty Source Window if there is no filename entered; however, in our example we entered 'pascal.x' as our cross-executable (Figure6.2).

Figure 6.2. Source Window



The menu selections in the Source Window are File, Run, View, Control, Preferences and Help. See 'Source Window Menus and Display Features' on following pages for more descriptions of the Source Window. To work with the other windows for debugging purposes specific to your project, use the View menu or the buttons in the toolbar (Figure 6.7).

If we had not initially specified a cross-executable file to open, we would now select a specific file by clicking on 'File|Open' in the Source Window. The file's contents will then be passed to the GDB interpreter for execution. To start debugging, click the 'Run' button (Figure 6.3) from the Source Window.

Figure 6.3. Run button

When the debugger runs, the button turns into the Stop button (Figure 6.4). If not connected to a target debug agent (i.e., serial, TCP, etc.), you can specify the 'Simulator', if your toolsuite has one, to run the program.

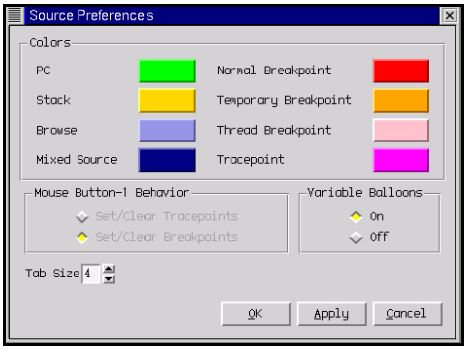
Figure 6.4. Stop button

The ‘Stop’ button interrupts the debugging process for a project, provided that the underlying hardware and protocols support such interruptions. Generally, machines that are connected to boards cannot interrupt programs on those boards, so the ‘Stop’ button has no functionality (it will appear unavailable, or “grayed out”). For more information on the toolbar buttons, see Figure 6.7.

WARNING: When debugging a target, do not click on the ‘Run’ button during an active debugging process, or it will de-activate the process. The ‘Run’ button will become the ‘Stop’ button and Visual GDB will lose connection with the target.

To specify preferences of how source code appears and to change debugging settings, select ‘Preferences|Source’ from the Source Window. The Source Preferences dialog opens (Figure 6.5).

Figure 6.5. Source Preferences Dialog



Left-click any of the colored squares to open the ‘Choose color’ dialog, with which you can modify the display colors of the Source Window. ‘Mouse Button-1 Behavior’ sets and clears either breakpoints or tracepoints (points in the source code, with an associated text string); the default is for setting breakpoints. ‘Variable Balloons’ lets you display a balloon of text whenever the cursor is over a variable in the Source Window; the balloon displays the value of the variable (see Figure 6.11 for an example). ‘On’ is the default selection. Selecting ‘Tab Size’ sets the number of spaces for a tab character in the Source Window. The Source Window has the following functionality and display features when using the Source Preferences dialog settings.

- When the executable is running in a debugging process, the location of the current program counter displays as a line with a colored background (PC).

- When the executable has finished running, the background color changes (Browse).

When looking at a stack backtrace, the background color changes to another different color (Stack). To set other preferences for a debugging session, select ‘Preferences | Global’ from the Source Window. The Global Preferences dialog opens (Figure 6.6) where you select a specific font and type size for the text in the windows for Visual GDB.

Figure6.6. Global Preferences Dialog

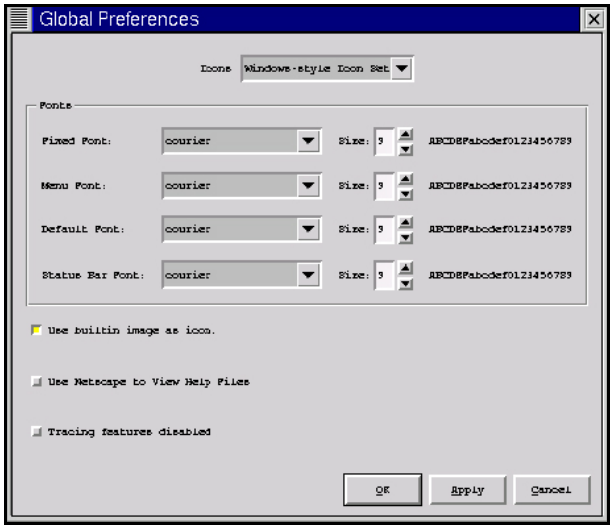
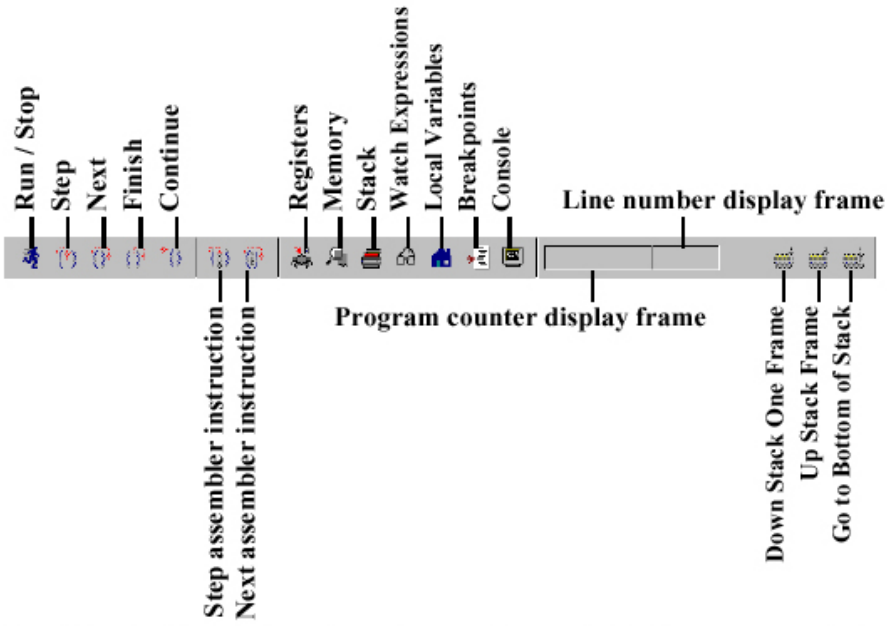


Figure 6.6 Global Preferences dialog icons allows you to select the appearance of the toolbar buttons as the Windows-style icon Set (the default; see Figure 6.7) or the Basic Icon Set (see Insight's Help menu for more information).

- Fonts is for selecting font family and size.
- Fixed Font sets the font for the source code display panes.
- Default Font sets the default font for list boxes, buttons and other controls.
- Status Bar Font sets the font for the status bar.
- Tracing features disabled disables setting tracepoints.

Figure 6.7. Toolbars / Menu



The following descriptions discuss the use of the default debugger toolbar buttons.

The **Run** button starts the debugging process for an executable file. If there is no executable open, the Load New Executable dialog displays to open an executable

During the debugging process, the **Run** button turns into the **Stop** button to interrupt the debugging. You cannot interrupt some targets; you will instead have to disconnect from the target.

The **Step** button steps to next executable line of source code. Also, the **Step** button steps into called functions.

The **Next** button steps to the next executable line of source code in the current file. Unlike the **Step** button, the **Next** button steps over called functions.

The **Finish** button finishes execution of a current frame. If clicked while in a function, it finishes the function and returns to the line that called the function.

The **Continue** button continues execution until a breakpoint, watchpoint or exception is encountered, or until execution completes.

The **Registers** button invokes the **Registers** window for viewing or changing register properties for a program's content.

The **Memory** button invokes the **Memory** window for displaying and editing the state of memory and addresses.

The **Stack** button invokes the **Stack** window for displaying and navigating the current call stack, where each line represents a stack frame.

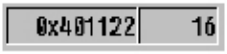
Figure 6.7. Toolbars / Menu (cont')

The **Watch Expressions** button invokes the **Watch Expressions** window for entering expressions which will be updated every time that the executable stops.

The **Local Variable** button invokes the **Local Variables** window for displaying all local variables and their structure.

The **Breakpoints** button invokes the **Breakpoints** window for examining breakpoints and changing their settings.

The **Console** button invokes the **Console** window as a command line interface for debugging. `(gdb)` is the prompt.



The left-hand read-only frame displays the program counter (`pc`) of the current frame.
The right-hand read-only frame displays the line number, which contains the program counter.

The **Step assembler** button steps through one assembler machine instruction. Also, the **Step assembler** button steps into subroutines.

The **Next assembler** button steps to the next assembler instruction. The **Next assembler** button then executes subroutines and steps to the next instruction.

The **Down Stack Frame** button moves down the stack frame one level.

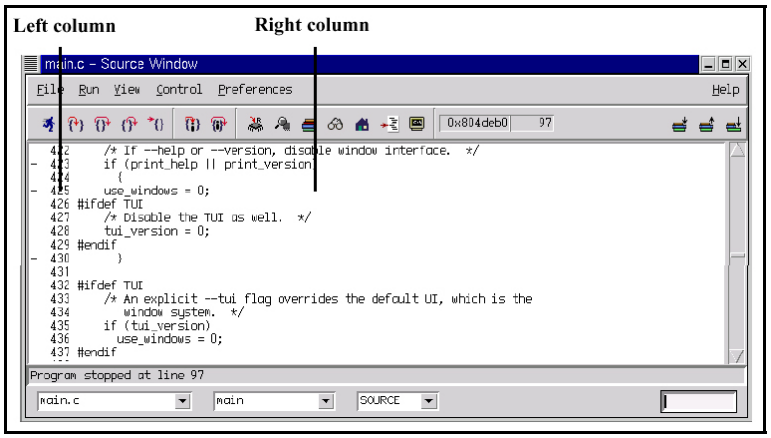
The **Up Stack Frame** button moves up the stack frame one level.

The **Go to Bottom of Stack Frame** button moves to the bottom of the stack frame.

6.3 Using the Mouse in the Source Window

The mouse has many uses within the main display pane of the Source Window. Divided into two columns (Figure 6.8), the window's left column extends from the left edge of the display pane to the last character of the line number, while the right column extends from the last character of the line number to the right edge of the display pane. Within each column, the mouse has different effects.

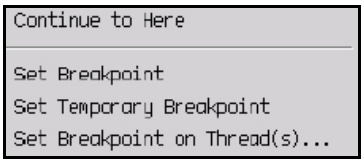
Figure 6.8. Using the Mouse in the Source Window



6.4 Left column functionality for the Source Window

When the cursor is in the left column over an executable line, it appears as a minus sign. When a breakpoint is set at this point, the cursor changes into a circle. A left click sets a breakpoint at the current line; the breakpoint appears as a colored square in place of the minus sign. A left click on any existing or temporary breakpoint removes that breakpoint. A right click on any existing or temporary breakpoint brings up a pop-up menu (Figure 6.9).

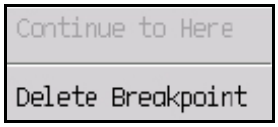
Figure 6.9. Pop-Up Menu for Setting Breakpoints



'Continue to Here' causes the program to run up to a location, ignoring any breakpoints; like the temporary breakpoint, this menu selection displays as a differently shaded square than a regular breakpoint. When a breakpoint has been disabled, it turns, for instance, from red or orange to black (color settings vary depending on the preferences you set; see also Figure 6.5 and its accompanying descriptions). 'Set Breakpoint' sets a breakpoint on the current executable line; this has the same action as left clicking on the minus sign. 'Set Temporary Breakpoint' sets a temporary breakpoint on a current executable line; a temporary breakpoint displays as a differently shaded square than a regular breakpoint, and is automatically removed when hit. 'Set Breakpoint on Thread(s)…' sets a thread-specific breakpoint at the current location.

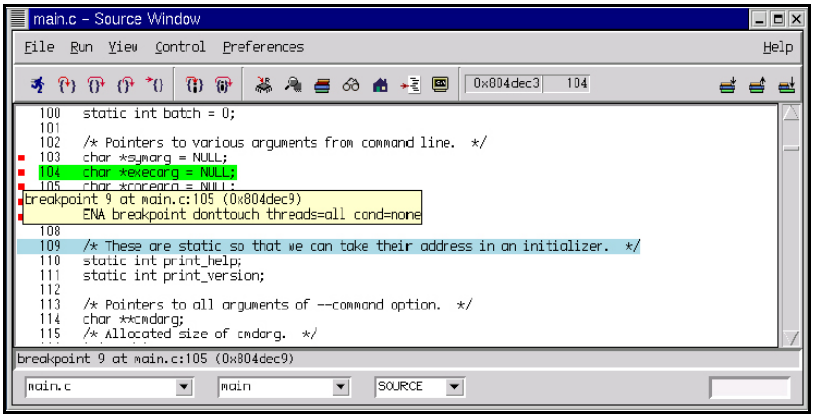
Right-click on a line with a breakpoint to invoke a pop-up menu to delete breakpoints (see Figure 6.10).

Figure 6.10. Pop-Up Menu for Deleting Breakpoints



Delete Breakpoint deletes the breakpoint on the current executable line. This has the same action as left clicking on the colored square; see the description for 'Continue to Here' for Figure 6.9. With the cursor over a line, a breakpoint opens a 'breakpoint information balloon'; see Figure 6.11 for an example of such a tool tip.

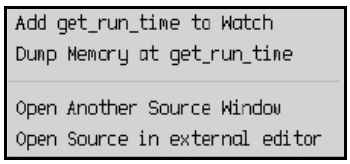
Figure 6.11. Breakpoint Information Balloon


main.c - Source Window
File Run View Control Preferences Help
0x804dec3 104
100 static int batch = 0;
101
102 /* Pointers to various arguments from command line. */
103 char *symarg = NULL;
104 char *execarg = NULL;
105 char *corearg = NULL;
Breakpoint 9 at main.c:105 (0x804dec9)
ENA breakpoint donttouch threads=all cond=none
108
109 /* These are static so that we can take their address in an initializer. */
110 static int print_help;
111 static int print_version;
112
113 /* Pointers to all arguments of --command option. */
114 char **cmdarg;
115 /* Allocated size of cmdarg. */
breakpoint 9 at main.c:105 (0x804dec9)
main.c
main
SOURCE

6.5 Right Column Functionality for the Source Window

The following documentation discusses the functionality of how the mouse works in the right column of the Source Window. With the cursor over a global or local variable, the value of that variable displays. With the cursor over a pointer to a structure or class, view the type of structure or class and the address of the structure or class. Double clicking an expression selects it. Right clicking an expression invokes a pop-up menu (see Figure 6.12).

Figure 6.12. Pop-Up Window for Expressions



'Add <selected expression>' to Watch opens the Watch Expressions window ('<selected expression>' in the example was 'get_run_time') and adds a variable expression to the list of expressions in the window. Dump Memory at '<selected expression>' opens the Memory window, which displays a memory dump at an expression. Open Another Source Window opens another Source Window for displaying a program in an alternate format (see Figure 6.16 and its accompanying descriptions). Open Source in external editor opens the program in an alternate editor, such as Xemacs.

6.6 Source Window Menus and Display Features

The Source Window has the following menu items, many of which correspond to the toolbar buttons.

- '**File**' has the following menu items:
  - 'Edit Source' allows direct editing of the source code.
  - 'Open' invokes the 'Load New Executable' dialog.
  - 'Source' invokes the 'Choose GDB Command File' dialog.
  - 'Exit' closes the Visual GDB program.

- '**Run**' has the following usage. 'Attach to Process' attaches thread processes for debugging (see 'Using the Processes Window for Threads' in this Section). Download downloads an executable to a target. Run runs the executable.
- '**View**' displays the following windows: 'Stack' (Figure 6.20), 'Registers' (Figure 6.21), 'Memory' (Figure 6.22), 'Watch Expressions' (Figure 6.24), 'Local Variables' (Figure 6.29), 'Breakpoints' (Figure 6.33), 'Console' (Figure 6.38), 'Function Browser' (Figure 6.39), and 'Processes' (for threads, use the 'Threads List' menu item).
- '**Control**' has the following usage. 'Step' steps to next executable line of source code and steps into called functions. 'Next' steps to next executable line of source code in the current file and steps over called functions. 'Finish' finishes execution of a current frame and, if clicked while in a function, finishes the function and returns to the line that called the function. 'Continue' continues execution until a breakpoint, watchpoint or exception is encountered, or until execution completes. 'Step Asm Inst' steps through one assembler machine instruction and steps into subroutines. 'Next Asm Inst' steps to the next assembler instruction but steps over subroutines.
- '**Preferences**' has the following usage. 'Global' opens 'Global Preferences' (Figure 6.6) for changing how text appears. 'Source' opens the 'Source Preferences' (Figure 6.5) to show how colors display.
- '**Help**' has the following usage. 'Help' displays the 'Help' window (Figure 6.42). 'About Visual GDB' displays the version number, copyright notices for Visual GDB.

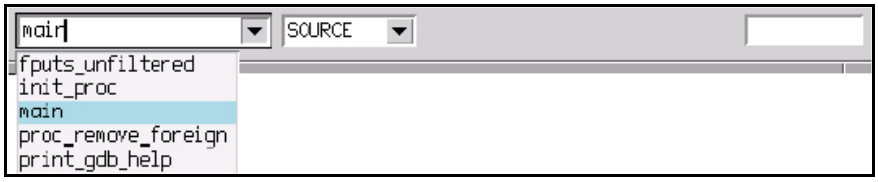
6.7 Below the Horizontal Scroll Bar of the Source Window

There are four display and selection fields below the horizontal scroll bar: the status text box (Figure 6.13), the file drop-down combo box (Figure 6.15), the function drop-down combo box (Figure 6.14) and the code display drop-down list box (Figure 6.16). At the top of the horizontal scroll bar, text details the current status of the debugger; the status text box in Figure 6.13 shows 'program stopped at line 19' as the message. The 'Function Browser' window provides even more powerful tools for locating files and functions within your source code; for more information.

Figure 6.13. Status Text Box

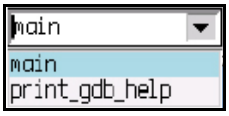
The function drop-down list box (Figure 6.14) displays all the functions of a selected source (.c) or header (.h) file that an executable uses. Select a function by clicking in the list, or by typing directly into the text field for the function drop-down list box.

Figure 6.14. Function Drop-Down Combo Box



The file drop-down list box (Figure 6.15) displays the source (.c) and header (.h) files associated with an executable. Select files by clicking the arrow to the right of the drop-down list and then selecting one of the files in the list, or by typing the file's name directly into the list's text field.

Figure 6.15. File Drop-Down List Box



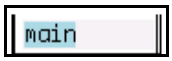
Select how the code in the source Window displays by using the code display drop-down list box (Figure 6.16).

Figure 6.16. Code Display Drop-Down List Box

The selections in the code display drop-down list box provide the following different ways to display code in the Source Window.

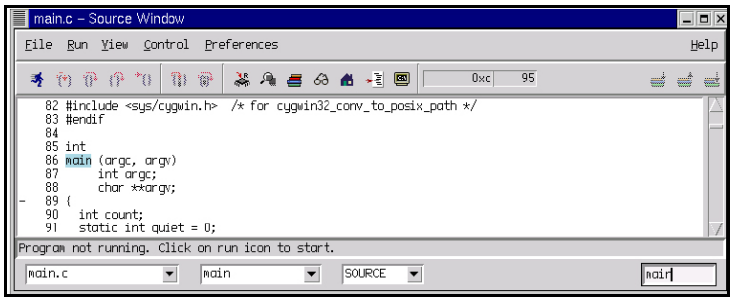
- SOURCE displays source code.
- ASSEMBLY displays assembly code.
- MIXED displays both source code and assembly code, interspersed within the Source Window.
- SRC+ASM displays a program's source and assembly code in separate panes.
- Type a character string into the search text box (Figure 6.17). Press Enter to perform a forward search on the source file for the first instance of a specific character string.

Figure 6.17. Search Text Box



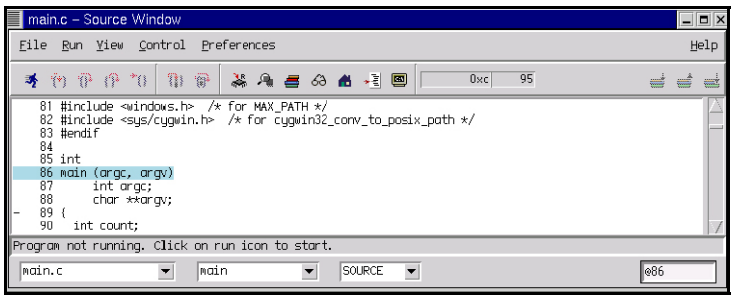
After having specified 'main' in the search text box, the example program in Figure 6.18 shows the jump to a main function.

Figure 6.18. Searching for a Word in Source Code



Use the Shift and Enter keys simultaneously to search for the string. Use the Enter key or the Shift and Enter keys to repeat the search. Type '@' with a number in the search text box and press Enter to jump to a specific line number in the source code. The example program in Figure 6.19 shows a jump to the line 86.

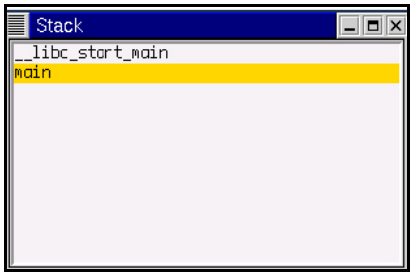
Figure 6.19. Searching for a Specific Line in Source Code



6.8 Using the Stack Window

Each time your program performs a function call, information about the call generates. That information includes the location of the call in your program, the arguments of the call, and the local variables of the function being called. The information is saved in a block of data called a *stack frame*. The stack frames are allocated in a region of memory called the *call stack*. When your program stops, you can examine the stack you to see this information. A stack refers to the layers (TCP/IP and sometimes others) through which all data passes at both client and server ends of a data exchange. The call stack is the data area or buffer used for storing requests that need to be handled, as in a list of tasks or, specifically, the contiguous parts of the data associated with one call to a specified function in a frame. The frame contains the arguments given to the function, the function's local variables, and the address at which the program is executing. The Stack window displays the current state of the call stack (Figure 6.20), where each line represents a stack frame; the line with the 'main.c' executable had been selected for the example.

Figure 6.20. Stack Window

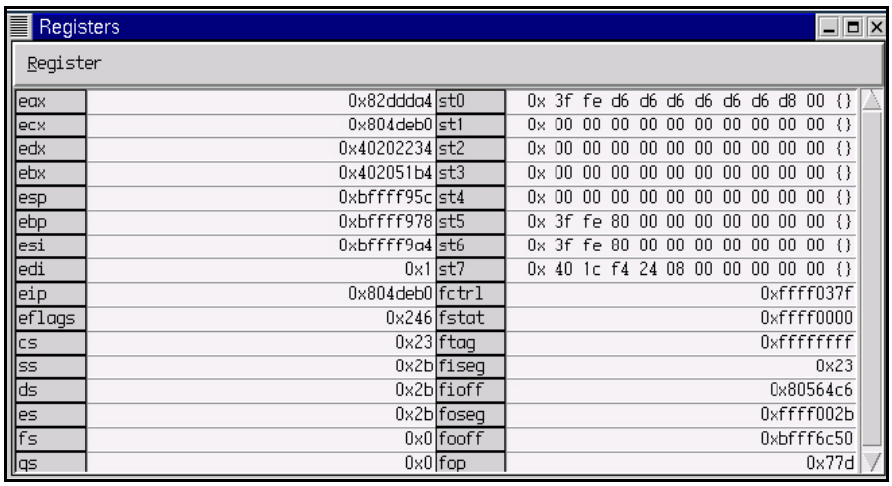


Click a frame to select or highlight that frame. The source window automatically shows the source code that corresponds to the selected frame. If the frame points to an assembly instruction, the source window changes to assembly code; the corresponding source line's background in the source window also changes to the stack color.

6.9 Using the Registers Window

The 'Registers' window (Figure 6.21) dynamically displays registers and their contents.

Figure 6.21. Registers Window



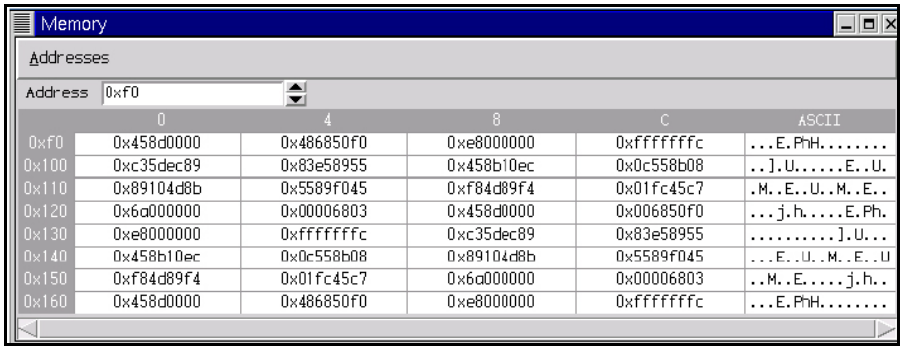
To change the properties of registers, use the following methods.

- To select a register, single left click on it.
- To edit the contents of a register, double click on it. Alternatively, use Register 'Edit' to change the contents after selecting a register. Use the 'Esc' key to abort the editing.
- Use Register 'Format' to invoke another pop-up menu to display the contents of a selected register in Hex (Hexadecimal), Decimal, Natural, Binary, Octal, or Raw formats. Hex is the default display format. Natural format refers to and Raw refers to the source format. The other formats are self-explanatory.
- Use Register 'Remove from Display' to remove a selected register from the window; all registers will display if you close and reopen the window, unless you have already selected this feature.
- Use Register 'Display All Registers' to display all the registers; this menu item is only active when one or more registers have been removed from display.

6.10 Using the Memory Window

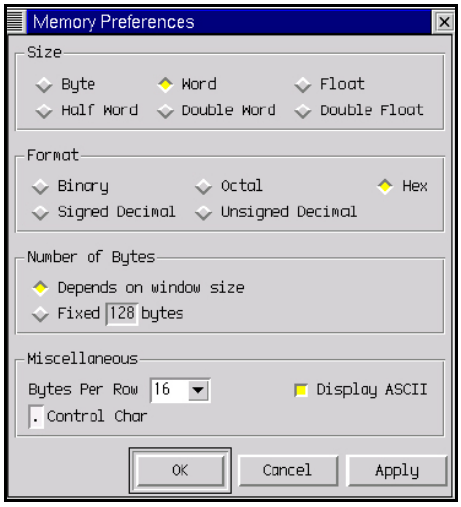
The Memory window (Figure 6.22) dynamically displays the state of memory. Double-click a memory location with the cursor in the window and edit its contents.

Figure 6.22. Memory Window



Use Addresses 'Auto Update' to update the contents of the Memory window automatically whenever the target's state changes; this is the default setting. Use Addresses 'Update Now' to update the Memory window's view of the target's memory.

Figure 6.23. Memory Preferences Dialog for the Memory window

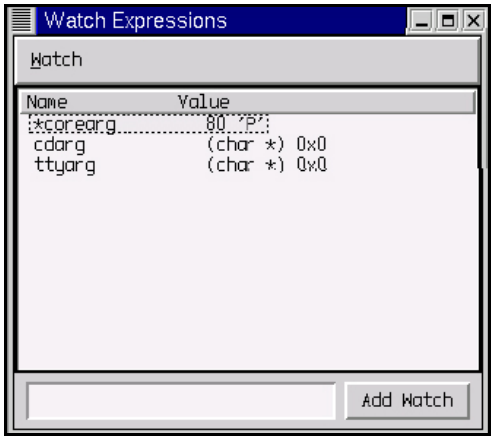


- Use Addresses 'Preferences' to invoke the Memory Preferences dialog to set memory options.
- Select the size of the individual cells to display with Size options; Byte, Half-Word, Word, Double-Word, Float, or Double-Float are the settings, with Word being the default selection.
- Select the format of the memory that displays with Format options; Binary, Signed Decimal, Octal, Unsigned Decimal, or Hex (Hexadecimal) are the settings, with Hex being the default selection.
- Set the number of bytes to display with Number of Bytes, Depends on Window Size or Fixed. Depends on Window Size selection is default.
- Display a string representation of memory with Miscellaneous, Bytes Per Row or Display ASCII selections. Control Char displays non-ASCII characters; the default control character is the period (.).

6.11 Using the Watch Expressions Window

The Watch Expressions window displays the name and current value of user-specified expressions (Figure 6.24).

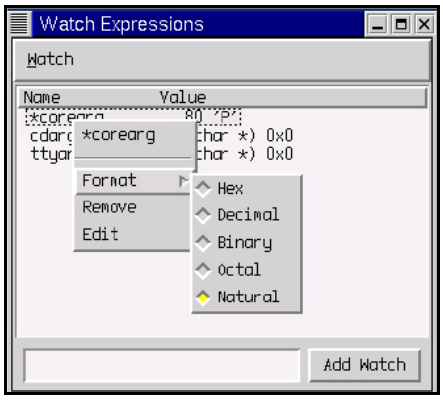
Figure 6.24. Watch Expressions Window



The Watch Expressions window has the following functionality.

- Single click on an expression to select it.
- Right click in the display pane, having selected an expression, to invoke an expression-specific Watch menu (Figure 6.25).

Figure 6.25. Watch Menu in the Watch Expressions Window



Use Watch 'Edit' to edit the value in an expression (an example of an expression capable of being edited is shown in Figure 6.26). Use the Esc key to abort editing.

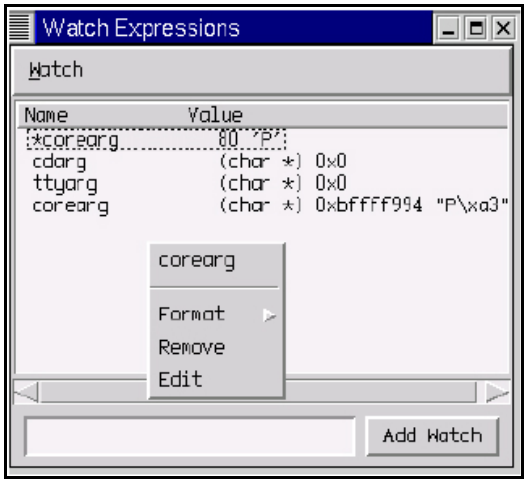
Figure 6.26. Editing the Value in an Expression

Use Watch Format to invoke another pop-up menu for displaying a selected expression's value in Hex (Hexadecimal), Decimal, Binary, or Octal formats; by default, pointers display in hexadecimal with all other expressions as decimal. Use Watch Remove to remove a selected expression from the watch list. Use the text edit field and the Add Watch button at the bottom of the window to add registers to the Watch Expression window or, by typing register convenience variables into the text edit field, add an expression to the watch list (see corearg added in Figure 6.27 with its results in Figure 6.28).

Figure 6.27. Using the Add Watch Button for the Watch Expressions Window

Every register has a corresponding convenience variable. The register convenience variables consist of a dollar sign followed by the register name; '$pc' is the program counter's convenience variable, for example, while '$fp' is the frame pointer's convenience variable. Re-cast other types to which a pointer was cast by typing it in the text edit field. For example, typing '(struct _foo *)' bar in the text edit field, the bar pointer is cast as a 'struct _foo' pointer. Invalid expressions are ignored.

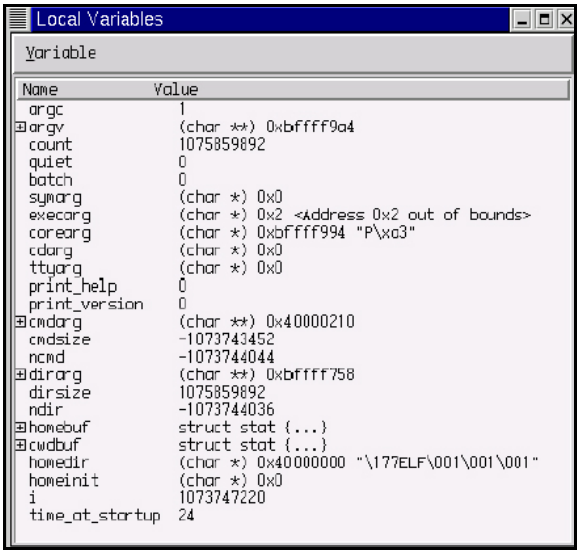
Figure 6.28. Results of Using Add Watch Button for the Watch Expressions Window



6.12 Using the Local Variables Window

The Local Variables window (Figure 6.29) displays the current value of all local variables.

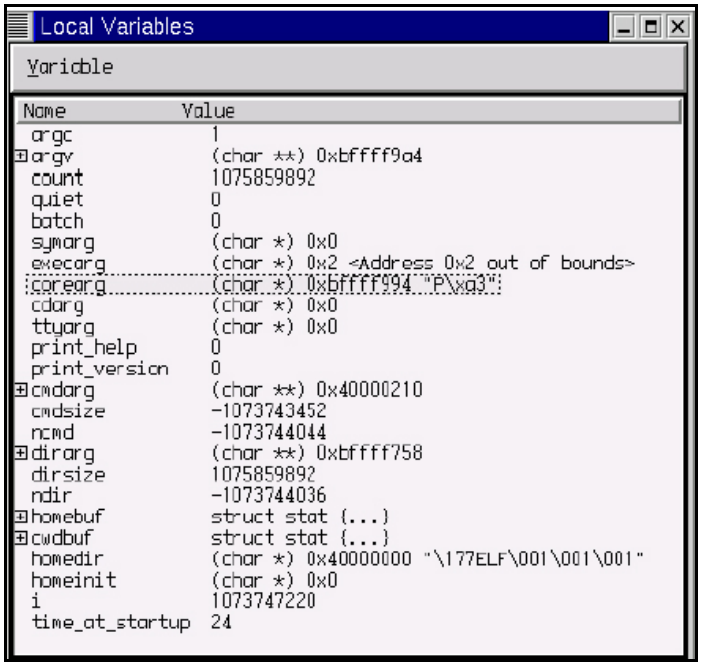
Figure 6.29. Local Variables Window



Use Variable 'Edit' to change the value of a selected variable that you want edit. Using the Escape key ('Esc') aborts editing. Use Variable 'Format' to invoke another pop-up menu to display a selected variable's value in Hex (Hexadecimal), Decimal, Binary or Octal formats. By default, pointers display in hexadecimal and all other

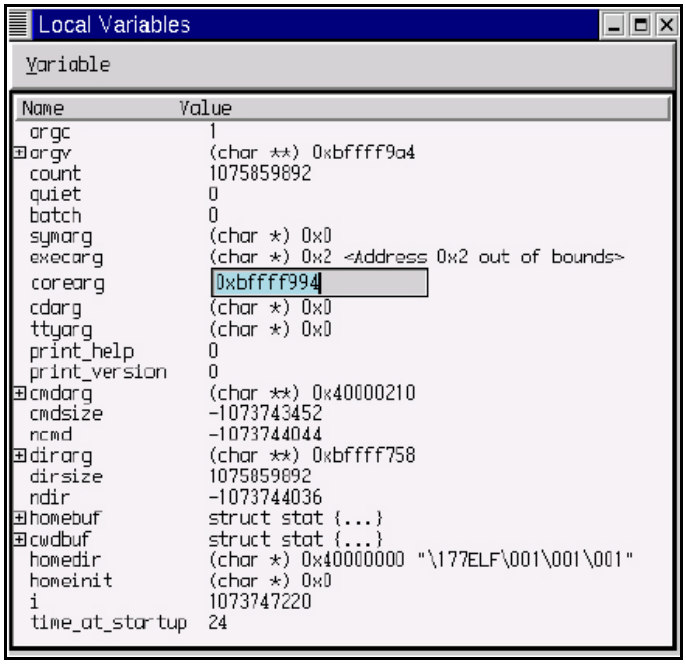
expressions as decimal. Single click the mouse with the cursor over a variable in the Local Variables window to select the variable (Figure 6.30).

Figure 6.30. Selecting a Variable



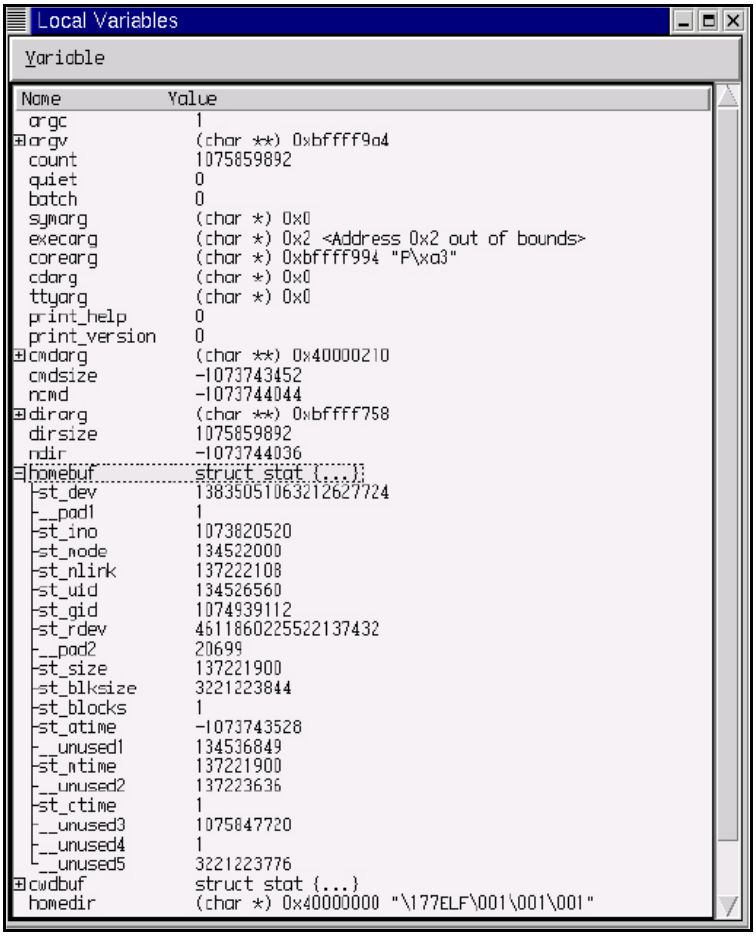
Double click the mouse with the cursor in the Local Variables window to edit a variable (Figure 6.31).

Figure 6.31. Editing Local Variables



Single click the mouse with the cursor on the plus sign to the left of a structure variable to see the elements of that structure (compare the variable structure for homebuf in Figure 6.30 with the results in Figure 6.32). To close the structure elements, click the minus sign to the left of an open structure (compare the variable structure in Figure 6.32 with what the window had displayed in Figure 6.30).

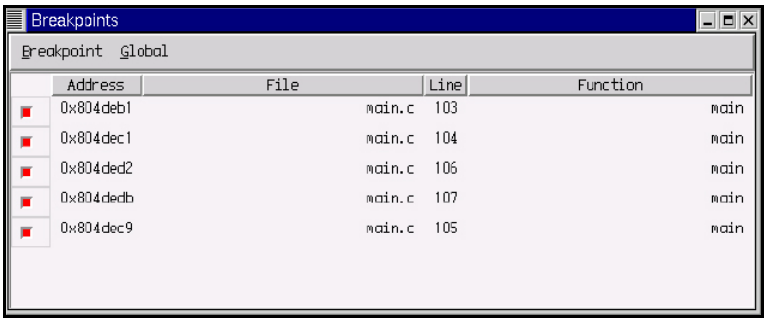
Figure 6.32. Displaying the Elements of a Variable Structure



6.13 Using the Breakpoints Window

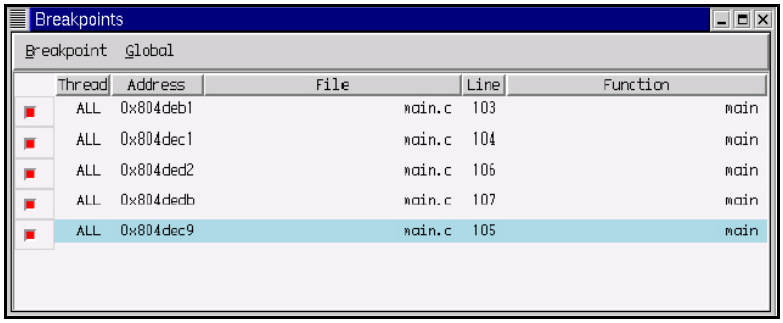
The Breakpoints window displays the currently set breakpoints. See Figure 6.33 for the ‘main.c’ example program breakpoints running in the Source Window, and see Figure 6.36 for the results in the Source Window. **WARNING:** Breakpoints and exceptions may not work, especially if debugging C++ code, and the Breakpoints window may be inoperative.

Figure 6.33. Breakpoints Window



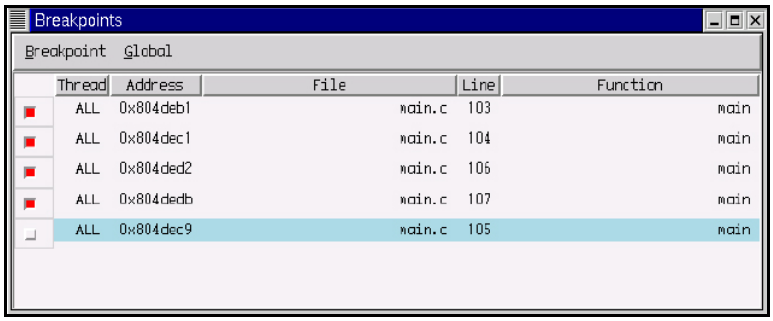
Single click the mouse with the cursor over a check-box for a breakpoint to select that breakpoint (see the breakpoint results in Figure 6.34).

Figure 6.34. Selecting a Breakpoint


Breakpoints
Breakpoint Global
Thread Address File Line Function
ALL 0x804deb1 main.c 103 main
ALL 0x804dec1 main.c 104 main
ALL 0x804ded2 main.c 106 main
ALL 0x804dedb main.c 107 main
ALL 0x804dec9 main.c 105 main

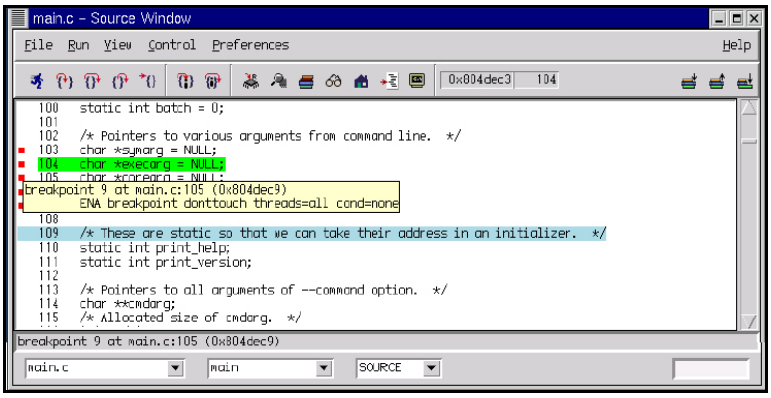
Single click with the mouse with the cursor over a check-box of a breakpoint to disable the breakpoint. The color of the square in the Breakpoint window changes (Line 101 in Figure 6.35) and the line's breakpoint status changes in the Source Window.

Figure 6.35. Setting Temporary Breakpoints in the Breakpoints Window


Breakpoints
Breakpoint Global
Thread Address File Line Function
ALL 0x804deb1 main.c 103 main
ALL 0x804dec1 main.c 104 main
ALL 0x804ded2 main.c 106 main
ALL 0x804dedb main.c 107 main
ALL 0x804dec9 main.c 105 main

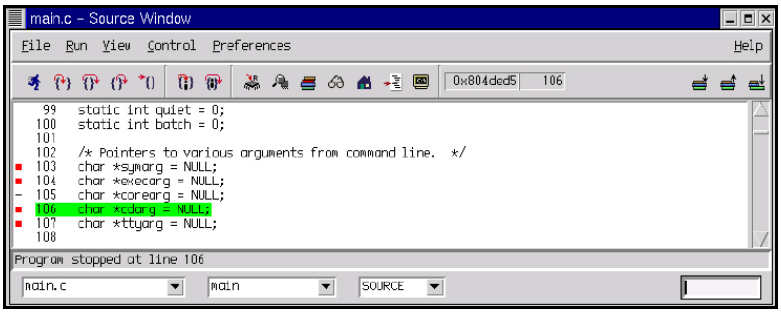
Using the Breakpoint menu for the Breakpoints window, toggle the enabled or disabled state of a selected breakpoint. The single check mark between them shows the state of the selected breakpoint. Remove removes the selected breakpoint. Using the Global menu for the Breakpoints window, Disable All disables all breakpoints, Enable All enables all breakpoints, and Remove All removes all breakpoints. Single click an empty check box of a disabled breakpoint to re-enable a breakpoint (Figure 6.36). A check reappears and the color of the square in the Source Window changes (see Figure 6.37).

Figure 6.36. Results in Source Window Having Enabled a Breakpoint


main.c – Source Window
File Run View Control Preferences Help
0x804dec3 104
100 static int batch = 0;
101
102 /* Pointers to various arguments from command line. */
103 char *symarg = NULL;
104 char *execarg = NULL;
105 char *corearg = NULL;
breakpoint 9 at main.c:105 (0x804dec9)
ENA breakpoint donttouch threads=all cond=none
108
109 /* These are static so that we can take their address in an initializer. */
110 static int print_help;
111 static int print_version;
112
113 /* Pointers to all arguments of --command option. */
114 char **cmdarg;
115 /* Allocated size of cmdarg. */
breakpoint 9 at main.c:105 (0x804dec9)
main.c
main
SOURCE

Using the Breakpoint menu, toggle between the normal and temporary setting of a selected breakpoint. A normal breakpoint remains valid no matter how many times it is hit. A temporary breakpoint is removed automatically the first time it is hit. A single check mark for either setting shows the state of the selected breakpoint. When a breakpoint is set to temporary, the line in the Source Window no longer has a colored square, as shown by comparing Figure 6.36 with Figure 6.37.

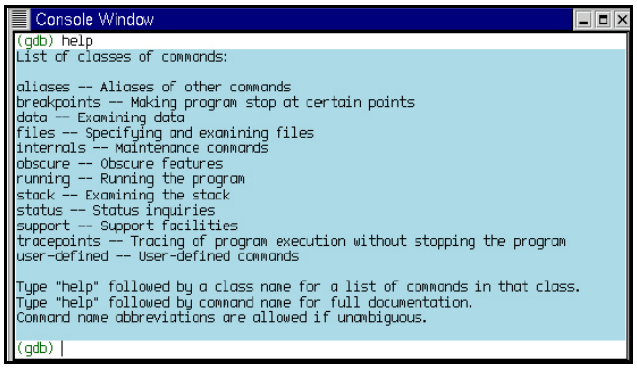
Figure 6.37. Results in Source Window Having Set a Breakpoint as Temporary


main.c – Source Window
File Run View Control Preferences Help
0x804ded5 106
99 static int quiet = 0;
100 static int batch = 0;
101
102 /* Pointers to various arguments from command line. */
103 char *symarg = NULL;
104 char *execarg = NULL;
105 char *corearg = NULL;
106 char *cdarg = NULL;
107 char *ttyarg = NULL;
108
Program stopped at line 106
main.c
main
SOURCE

6.14 Using the Console Window

To send commands directly to the GDB interpreter, use the Console window (Figure 6.38).

Figure 6.38. Console window

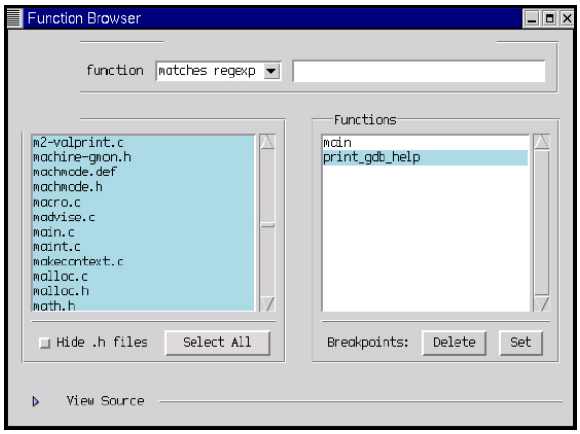

Console Window
(gdb) help
List of classes of commands:
aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands
Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)

The Console window opens with a (GDB) prompt for invoking debugging commands. Figure 6.38 shows the help command's available topics when using the Console window. For more specific commands, see *MC-Debugging-Tools.pdf* in the docs directory of the i.MX GNU X-Tools CD or in the docs directory under Cygwin.

6.15 Using the Function Browser Window

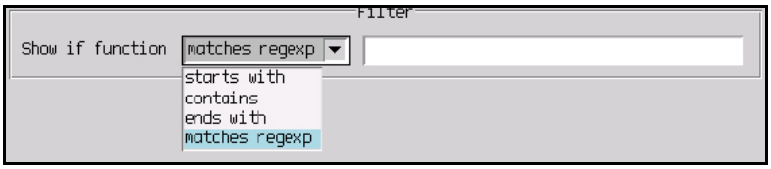
To invoke the Function Browser window, select 'View|Function Browser' from the Source Window. The Function Browser window has several fields that provide search and browsing capability for source code debugging (Figure 6.39). Descriptions follow of the Filter, Files, Functions and View Source fields.

Figure 6.39. Function Browser Window



The Filter group at the top of the Function Browser window contains the Show if function drop-down list box and a text edit field. Show if function allows you to match the character string in the text edit field to its right by any of the four alternatives. Using the Show if function drop-down list box (Figure 6.40), starts with shows functions that start with the character string in the text edit field entry, contains shows functions that contain the character string in the text edit field entry, ends with shows functions that end with the character string in the text edit field entry, matches regexp makes the search routines use regular expression matching (for example, searching for '^[ab].*' matches all functions starting with either a or b letters).

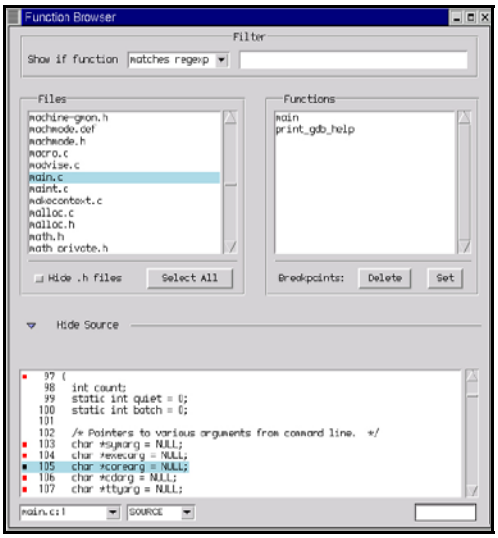
Figure 6.40. Show if Function Drop-Down List Box



The Files group limits the search to highlighted files. Click individual file names to select or deselect that file. The list of matching files refreshes when any search parameter changes. Hide '.h' files, if checked, disallows '.h' header files to display.

Select All selects all listed files. The Functions group matches all functions in the selected file(s). Breakpoints has two available buttons, Delete or Set; Delete removes a breakpoint previously set at the first executable line of the selected function, while Set sets a breakpoint at the first executable line of the selected function. Both of these will work on any and all selected functions in the listing. If all functions are selected, they all get or lose a breakpoint. View Source/Hide Source allows you to toggle between displaying or hiding a file in a source browser (Figure 6.41); the source browser has the same functionality as when using the Source Window.

Figure 6.41. Function Browser Window with Source Browser



There are four display and selection fields below the horizontal scroll bar (the same functionality as using the Source Window): the status text box (Figure 6.13), the function drop-down combo box (Figure 6.14) and the code display drop-down list box (Figure 6.16); see the figures and their accompanying explanations for specific information.

6.16 Using the Processes Window for Threads

The Processes window dynamically displays the state of currently running threads. **WARNING:** Threads support is not available for all targets. The Processes window will display a list of threads and/or processes of an executable that you are debugging. The exact contents are specific to each operating system. The first column is the thread number, used internally by the debugger to track the thread. This number is also used by the command line interface (in the Console window) when referring to threads. The rest of the columns are dependent on information coming from the operating system. The Source Window displays the current location and source for a current thread (or process). To change the current thread, click on the desired thread in the Processes window and the debugger will switch contexts, updating all windows. The current thread will highlight. Having set a breakpoint on a line or function, stop execution and return control to the debugger for every thread that hits a set location. To set a breakpoint on a specific thread or threads, use the Source Window. See also *Setting Breakpoints and Viewing Local Variables* and *Setting Breakpoints on Multiple Threads* in this Section.

6.17 Using the Help Window

Invoke the Help window (Figure 6.42) using the Help menu to get HTML-based navigable help by topic.

Figure 6.42. Help Window Showing the Help Topic's Index

The Help window has two menus: File and Topics. The File menu makes the following options functional: Back moves back one HTML help page, relative to previous forward page movements; Forward moves forward one HTML help page, relative to previous back page movement; Home returns to the main HTML help Table of Contents page; Close closes the Help window. The Topics menu displays information for each menu item. Content changes in the Help window to represent a selected topic. The first menu item, index, returns to the main Help window (Figure 6.42). The second item, Attach Dialog, is only for a host system's use, when attaching to another debugging process, and *not* for use by embedded targets. The remaining menus document the Insight windows: Stack (Figure 6.20), Registers (Figure 6.21), Memory (Figure 6.22), Watch Expressions (Figure 6.24), Local Variables (Figure 6.29), Breakpoints (Figure 6.33), Console (Figure 6.38), Function Browser (Figure 6.39), and Threads (for the Processes window when working with threads; the window contents are dependent on the operating system in use).

6.18 Examples of Debugging with Visual GDB

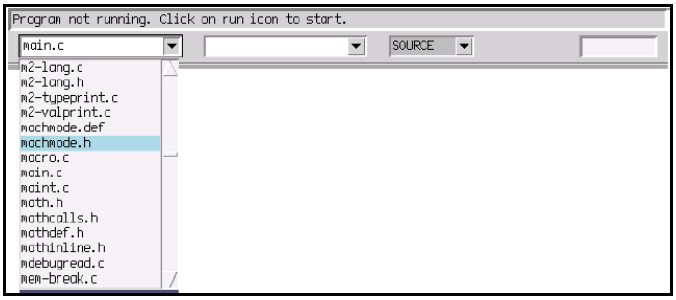
The following documentation contains examples of debugging session procedures for using Visual GDB; the content assumes familiarity with GDB and its main debugging procedures.

6.18.1 Selecting and Examining a Source File

To select a source file, or to specify what to display when examining a source file when debugging, use the following process.

1. Select a source file from the file drop-down list, at the bottom left of the Source Window ('main.c' in the example in Figure 6.43).

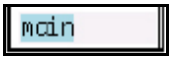
Figure 4.43. Source File Selection



2. Select a function from the function drop-down list to the right of the file drop-down list, or type its name in the text field above the list to locate a function.

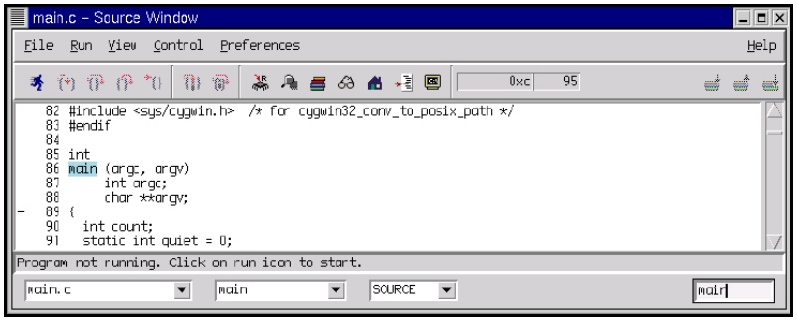
3. Type a character string into the search text box (Figure 4.44).

Figure 4.44. Search Text Box



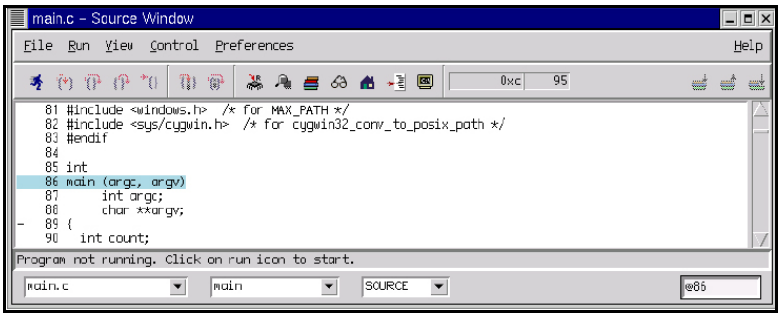
4. Press Enter to perform a forward search on the source file for the first instance of a specific character string. After having specified main in the search text box, the example program in Figure 6.45 shows the jump to a main function.

Figure 6.45. Searching for a Word in Source Code



5. Use the Shift and Enter keys simultaneously to search for the string. Use the Enter key or the Shift and Enter keys to repeat the search. Type '@' with a number in the search text box and press Enter to jump to a specific line number in the source code. The example program in Figure 4.46 shows a jump to the line 86.

Figure 6.46. Searching for a Specific Line in Source Code



Setting Breakpoints and Viewing Local Variables

A breakpoint can be set at any executable line in a source file. Executable lines are marked by a minus sign in the left margin of the Source Window. When the cursor is in the left column and it is over an executable line, it changes into a circle. When the cursor is in this state, a breakpoint can be set. The following exercise steps you through setting four breakpoints in a function, as well as running the program and viewing the changing values in the local variables.

1. With the Source Window active and the 'main.c' source file open, the cursor was placed over the minus sign on line 6.

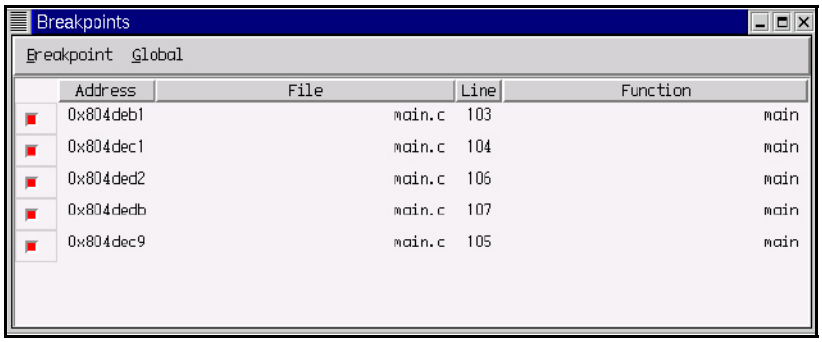
2. When the minus sign changes into a circle, click the left mouse button; this sets the breakpoint, indicated by a colored square.

3. Click on a breakpoint to remove the breakpoint.

4. Repeat the process to set breakpoints at specific lines.

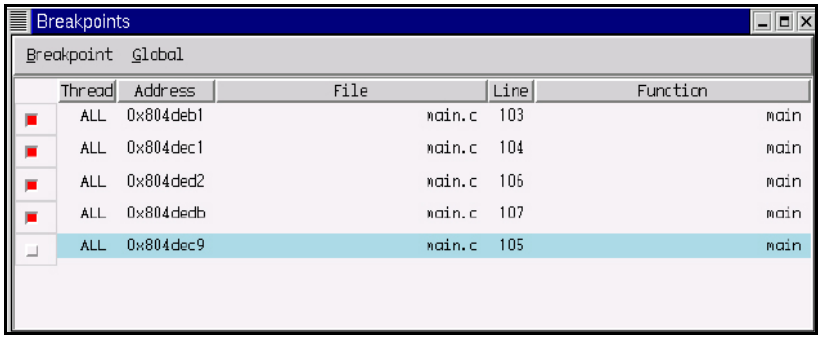
5. Open the Breakpoints window (Figure 6.47).

Figure 6.47. Breakpoints Window



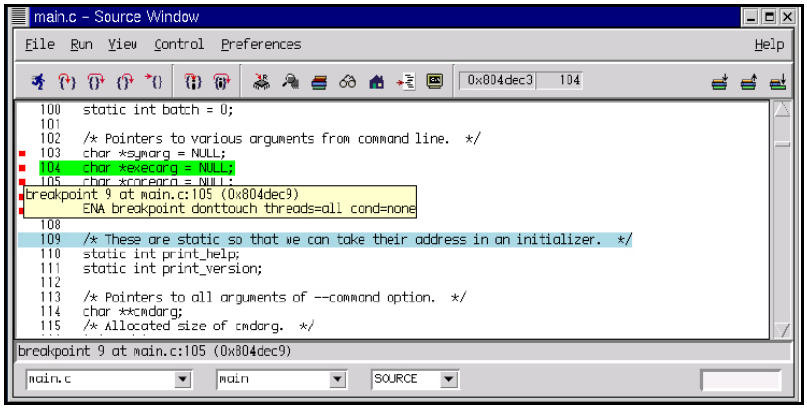
6. Click the check box for a line to set a breakpoint in an executable. The box's color changes and the square's color of the line in the Source Window changes (Figure 6.48). This color change indicates a disabling of the breakpoint. Re-enable the breakpoint at the line by clicking the check box in the Breakpoints window.

Figure 6.48. Disabling a Breakpoint in Breakpoints Window


Breakpoints
Breakpoint Global
Thread Address File Line Function
ALL 0x804deb1 main.c 103 main
ALL 0x804dec1 main.c 104 main
ALL 0x804ded2 main.c 106 main
ALL 0x804dedb main.c 107 main
ALL 0x804dec9 main.c 105 main

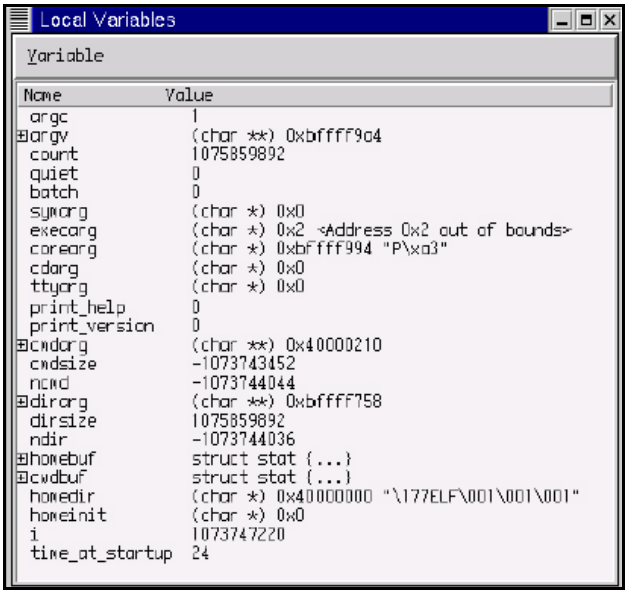
7. Click the Run button on the tool bar to start the executable. The program runs until it hits the first breakpoint. The color bar on the line changes color, indicating that the program is running (see settings in Figure 6.47 changed in Figure 6.48, and the Source Window in Figure 6.49 after debugging stopped).

Figure 6.49. Results of Setting Breakpoints at Line 105


main.c - Source Window
File Run View Control Preferences Help
0x804dec3 104
100 static int batch = 0;
101
102 /* Pointers to various arguments from command line. */
103 char *symarg = NULL;
104 char *execarg = NULL;
105 char *corearg = NULL;
breakpoint 9 at main.c:105 (0x804dec9)
ENA breakpoint donttouch threads=all cond=none
108
109 /* These are static so that we can take their address in an initializer. */
110 static int print_help;
111 static int print_version;
112
113 /* Pointers to all arguments of --command option. */
114 char **cmdarg;
115 /* Allocated size of cmdarg. */
breakpoint 9 at main.c:105 (0x804dec9)
main.c
main
SOURCE

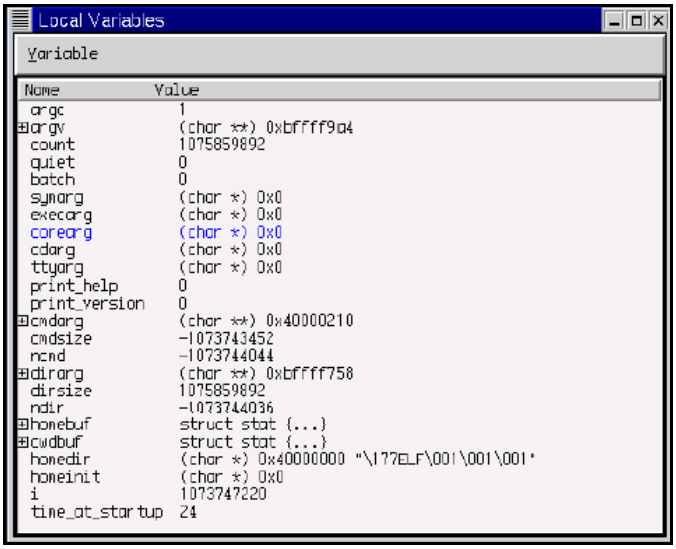
8. Open the Local Variables window (Figure 6.50), by clicking the Local Variables button on the toolbar. The window displays the initial values of the variables.

Figure 6.50. Local Variables Window



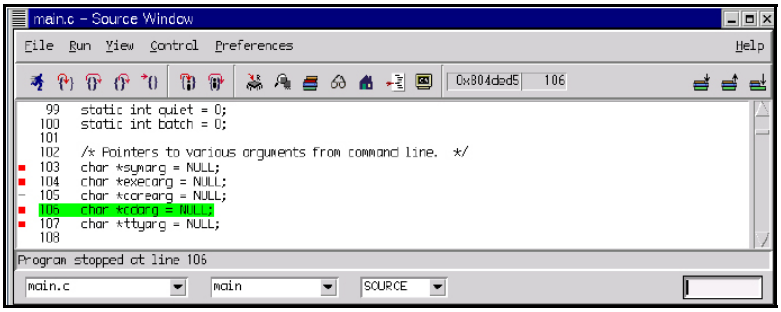
9. Click the Continue button in the tool bar to move to the next breakpoint. The variables that have changed value turn color in the Local Variables window (see results in Figure 6.51 for line 105 in the ‘main.c’ example).

Figure 6.51. Local Variables Window after Setting Breakpoints



10. Click the Continue button two more times to step through the next two breakpoints and notice that the values of the local variables change (compare results from the 'main.c' example program in Figure 6.49 and results in Figure 6.52). Repeat with the Continue button to step through breakpoints and notice their values change.

Figure 6.52. File after Changing Local Variables Values

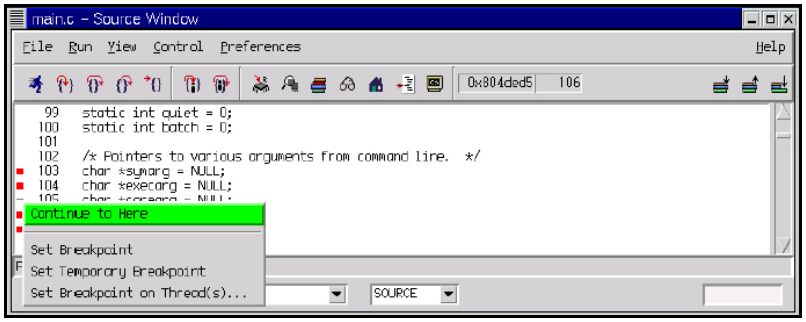

main.c – Source Window
File Run View Control Preferences Help
0x804dad5 106
99 static int quiet = 0;
100 static int batch = 0;
101
102 /* Pointers to various arguments from command line. */
103 char *symarg = NULL;
104 char *execarg = NULL;
105 char *corearg = NULL;
106 char *cdarg = NULL;
107 char *ttyarg = NULL;
108
Program stopped at line 106
main.c
main
SOURCE

6.18.2 Setting Breakpoints on Multiple Threads

With Visual GDB processing in a multi-thread environment, select threads and set breakpoints on one or more threads when debugging. **WARNING:** Multiple thread functionality does not work similarly on all embedded targets. When debugging C++ code, for instance, breakpoints and exceptions may not work on multiple threads. A process can have multiple threads running concurrently, each performing a different task, such as waiting for events or something time-consuming that a program doesn't need to complete before resuming. When a thread finishes its job, the debugger suspends or destroys the thread running in the debugging process. The thread debugging facility allows you to observe all threads while your program runs. However, whenever the debugging process is active, one thread in particular is always the focus of debugging. This thread is called the *current thread.* The precise semantics of threads and the use of threads differs depending on operating systems. In general, the threads of a single program are like multiple processes—except that they share one address space (that is, they can all examine and modify the same variables). Additionally, each thread has its own registers and execution stack, and perhaps private memory.

1. In the Source Window, right click on an executable line without a breakpoint to open the breakpoint pop-up menu (see Figure 6.53).

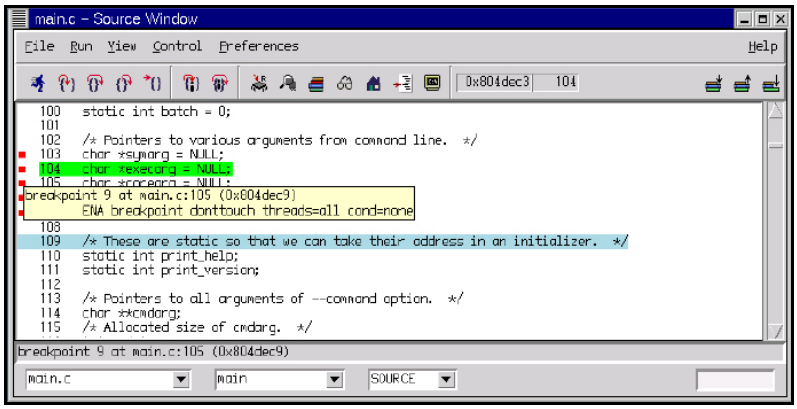
Figure 6.53. Breakpoint Pop-Up Menu in the Source Window


main.c - Source Window
File Run View Control Preferences Help
0x804ded5 106
99 static int quiet = 0;
100 static int batch = 0;
101
102 /* Pointers to various arguments from command line. */
103 char *symarg = NULL;
104 char *execarg = NULL;
Continue to Here
Set Breakpoint
Set Temporary Breakpoint
Set Breakpoint on Thread(s)...
SOURCE

2. Select the Set Breakpoint on Thread(s) menu item. The Processes window displays.

3. By clicking on specific breakpoints, select one or more threads. A breakpoint sets in the Source Window at the executable line only for the selected threads. Having selected threads, the results display in the Processes window. With the cursor over a breakpoint at line 105 in the sample program in the Source Window, a breakpoint information balloon displays to show where the selected thread begins (Figure 6.54).

Figure 6.54. Breakpoint Balloon with Thread Information in Source Window


main.c - Source Window
File Run View Control Preferences Help
0x804dec3 104
100 static int batch = 0;
101
102 /* Pointers to various arguments from command line. */
103 char *symarg = NULL;
104 char *execarg = NULL;
105 char *corearg = NULL;
breakpoint 9 at main.c:105 (0x804dec9)
ENA breakpoint donttouch threads=all cond=none
108
109 /* These are static so that we can take their address in an initializer. */
110 static int print_help;
111 static int print_version;
112
113 /* Pointers to all arguments of --command option. */
114 char **cmdarg;
115 /* Allocated size of cmdarg. */
breakpoint 9 at main.c:105 (0x804dec9)
main.c
main
SOURCE

This concludes our Section on how to use Visual GDB. More information may be gotten using the on-line help in Visual GDB.

6.19 Visual GDB with JTAG/BDM Debug Agents

Microcross is partnered with several JTAG/BDM tool vendors, including Abatron, American Arium, EPI Tools, Macraigor, Nohau and Signum. Each of these vendors support the GDB stub protocol and test their tools using the Microcross i.MX GNU X-Tools. In this section we present two examples how how to use Visual GDB with an Abatron BDI2000 and a Macraigor mpDemon.

6.19.1 Abatron BDI2000 Setup and Debug with Visual GDB

Introduction

Refer to Abatron's User Manual to install the CPU target firmware and configure on a network. Microcross will discuss tools specific information necessary to use Visual GDB, or for that matter command line GDB, and get up and going with the BDI2000 debug agent.

Debugging with GDB

Because the target runs within the BDI, no debug support has to be linked to your application. There is also no need for any BDI specific changes in the application sources. Your application must be fully linked because no dynamic loading is supported.

Target Setup

Target initialization may be done at two places. First with the BDI configuration file; second, within the application. The setup in the configuration file must at least enable access to the target memory where the application will be loaded. Disabling the watchdog timer and setting the CPU clock rate should also be done in the BDI configuration file. Application specific initializations like setting the timer rate are best located in the application startup sequence.

Steps to a Quick Setup

BDI2000 Simplified Configuration Steps for Board Support

1. Unzip the BDI2000 zip file of programs and configuration scripts that come with the BDI2000 into a directory on your local host hard drive – hereafter referred to as the **bdi2000** directory.
2. Connect the BDI2000 serial port connector to the BDI2000 and your host computer. You may connect your Ethernet cable to a hub or have a cross-over cable to directly connect to the host computer and BDI2000
3. Power-up the BDI2000.
4. Run a program called 'B20ARMGD.exe' that is in your **bdi2000** directory.
5. Click on the menu 'Setup|BDI2000'.
6. Click on the 'Connect' button and establish communications. Ensure you have a valid serial connection before proceeding.
7. Enter the configuration information:
    a. BDI IP Address – one you will assign that is not duplicated on your network.
    b. Subnet mask.
    c. Default Gateway.
    d. Config – Host IP Address – the host machine you have the configuration software running on along with where TFTP server program that resides in the **bdi2000 directory**.
8. You may check the status of your firmware by clicking on the 'Current' button. If out of date/version, click on the 'Update' button.
9. Save the configuration data into a file of your choosing using the 'File|Save As' in the menu.
10. Click on 'Transmit' to send this configuration to the BDI2000 via serial.
11. Exit the 'B20ARMGD.exe' application.
12. Edit the 'xxxxx.cfg' file. Modifying the following lines:
    a. Line where Host IP is located: Enter Host IP address where TFTP runs (from a DOS command shell, use 'ipconfig' to find out what the Host IP address is).
    b. Enter the REGS file path – path should be same as step 1 (bdi2000 directory).
13. Create a shortcut to 'tftpsrv.exe' that is located in the **bdi2000** directory and execute it to start the TFTP Server before going to the next step – very important.

14. Unplug power to the board and BDI2000, and then connect the JTAG connector to the development board.
15. Powerup the board and BDI2000.
16. Start a DOS command shell and enter the command:
    'telnet <IP-address-assigned-to-BDI2000>'
    The response should be a command prompt 'BDI>' or custom prompt if set in the configuration file (e.g., 'CSB536'>).
17. At this point you should be able to connect via GDB or Visual GDB using the appropriate TCP address and port number.

Connecting to the Target with the Abatron BDI2000

Command Line GDB

After starting your debugger with binary you created with debug symbols in it, you must issue commands to remotely connect to the target. This can be done by issuing the following commands:

$ <target-alias>-gdb <filename> (enter)
(you must be in the directory where the binary, <filename>, is located)

(GDB) target remote <bdi2000-TCP-Addr>:2001 (enter)

The <bdi2000-TCP-Addr> stands for an IP address assigned to the BDI2000. The host configuration (xxxx.cfg) file must have an appropriate entry. 2001 is the default TCP port used to communicate with the BDI.

Visual GDB Startup

1. Start the Visual GDB debugger with a binary built with debug symbols, <filename>, and click on 'File|Target Setting' and following the procedures below to completely setup communications within Visual GDB:

   $ <target-alias>-gdbtk <filename> (enter)
   (you must be in the directory where the binary, <filename>, is located; if using Visual X-Tools to start the Visual GDB debugger, then simply click on the icon or menu item under Build)

2. Click on 'Target' combo-box and select 'Remote/TCP'.

3. Enter the 'TCP/IP address' of the Abatron BDI2000.

4. Enter 'port address'— if setup as instructed, it should be '2001'.

5. Select the desired check boxes.

6. Click on 'More Options' enter any desired 'Run Options' and click 'OK'. If debugging on the hardware target, select the run option 'continue' since a program cannot be run in the traditional sense through the JTAG.

7. Set your breakpoints, if necessary, and click on 'run', and thereafter you can single step or continue. Click on 'View' to select view options like registers, memory, console, function browser, etc.. Review Section 5 for more details on how to use Visual GDB.

5.19.2 Macraigor mpDemon Setup and Debug with Visual GDB

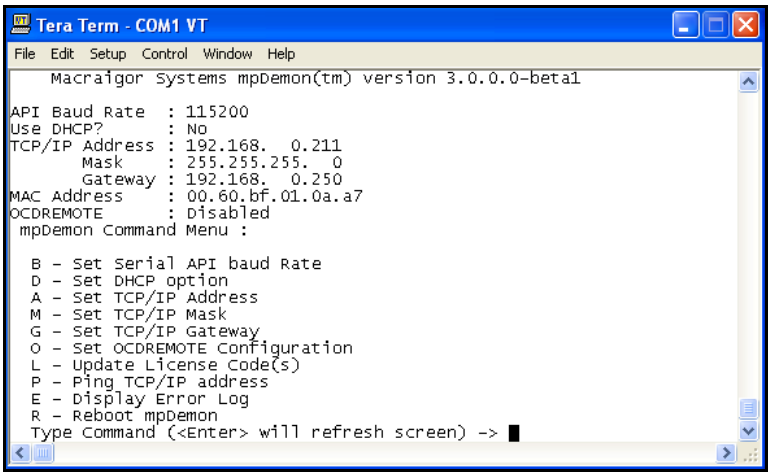
Introduction

The Macraigor mpDemon version 3.0.0.x can now communicate directly with one or more GDB sessions via Ethernet to it's internal OCDRemote GDB stub interface. Initially, prior to starting the Visual GDB or command line GDB session(s) on your host, you must first configure the mpDemon's OCDRemote for your target board and local network settings. OCDRemote converts GDB commands to JTAG/BDM signals that control your target board's CPU(s).

To configure your mpDemon you must connect it to your network, make a serial connection to a host computer, and follow a few simple steps to get up and running with Visual GDB. Follow these steps to quickly configure the mpDemon:

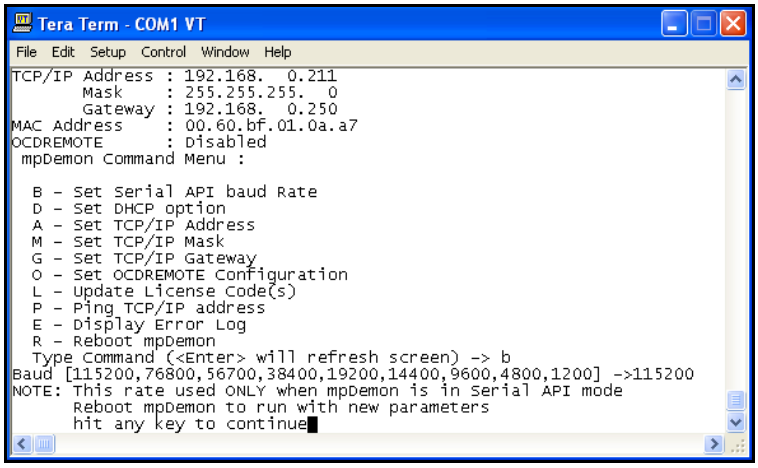
1. With power off, connect the appropriate JTAG/BDM header connector to the board and the other end to the mpDemon.
2. Connect the Macraigor supplied serial cable to the host computer's available serial port and the other end into the mpDemon. Connect the power cable to the mpDemon and target board. The mpDemon will show power indication on the front panel with an LED lit.
3. Start a Serial Terminal serial communications program.
4. Set the serial connection speed to 19,200 baud, no parity, 8 data bits, 1 stop bit and no flow control.
5. Press the Enter key on the keyboard, and you should get a boot setup screen showing up in your serial monitor program. Figure 6.55 shows the first boot screen.

Figure 6.55. Basic Boot Screen of mpDemon for Setup


Tera Term - COM1 VT
File Edit Setup Control Window Help
Macraigor Systems mpDemon(tm) version 3.0.0.0-beta1
API Baud Rate : 115200
Use DHCP? : No
TCP/IP Address : 192.168. 0.211
Mask : 255.255.255. 0
Gateway : 192.168. 0.250
MAC Address : 00.60.bf.01.0a.a7
OCDREMOTE : Disabled
mpDemon Command Menu :
B - Set Serial API baud Rate
D - Set DHCP option
A - Set TCP/IP Address
M - Set TCP/IP Mask
G - Set TCP/IP Gateway
O - Set OCDREMOTE Configuration
L - Update License Code(s)
P - Ping TCP/IP address
E - Display Error Log
R - Reboot mpDemon
Type Command (<Enter> will refresh screen) ->

6. Enter the letter 'B' to set the serial API baud rate. Enter a desired value from the list of options and press Enter. See Figure 6.56 for a screen shot of this option.

Figure 6.56. Set Serial API Baud Rate


Tera Term - COM1 VT
File Edit Setup Control Window Help
TCP/IP Address : 192.168. 0.211
Mask : 255.255.255. 0
Gateway : 192.168. 0.250
MAC Address : 00.60.bf.01.0a.a7
OCDREMOTE : Disabled
mpDemon Command Menu :
B - Set Serial API baud Rate
D - Set DHCP option
A - Set TCP/IP Address
M - Set TCP/IP Mask
G - Set TCP/IP Gateway
O - Set OCDREMOTE Configuration
L - Update License Code(s)
P - Ping TCP/IP address
E - Display Error Log
R - Reboot mpDemon
Type Command (<Enter> will refresh screen) -> b
Baud [115200,76800,56700,38400,19200,14400,9600,4800,1200] ->115200
NOTE: This rate used ONLY when mpDemon is in Serial API mode
Reboot mpDemon to run with new parameters
hit any key to continue

7. Ask your network administrator for details about your network configuration, so you can assign an IP address, gateway, network mask, or alternatively you can enter DHCP configuration by selecting 'D' in the mpDemon command menu and press Enter, and then enter 'Y' – see Figures 6.57 and 6.58. Next, enter the DHCP host name or IP address.

Figure 6.57. Use DHCP?

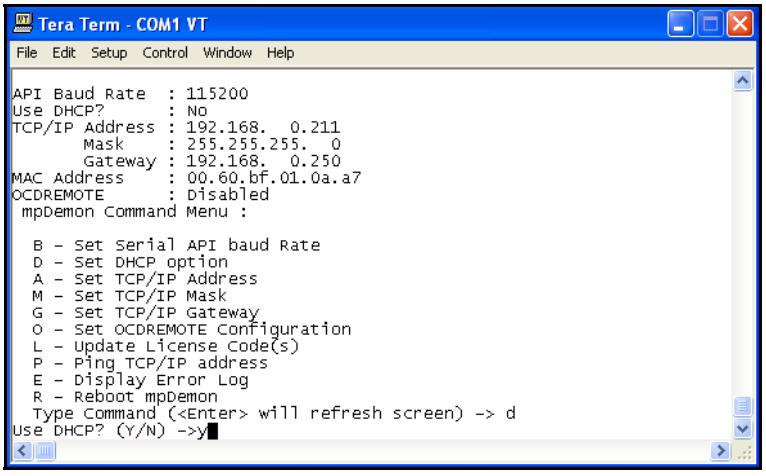
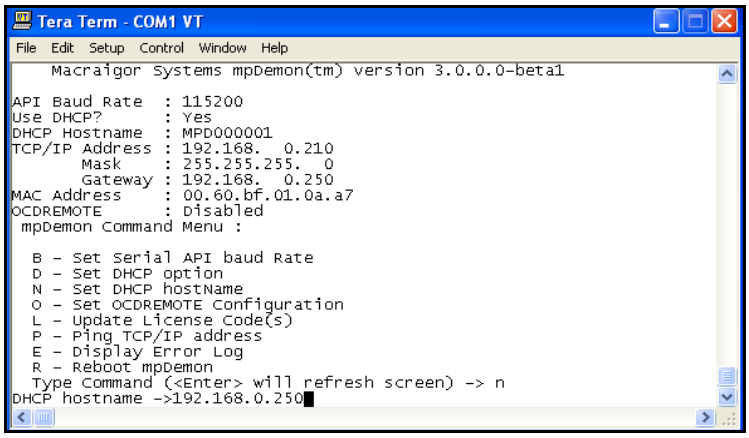
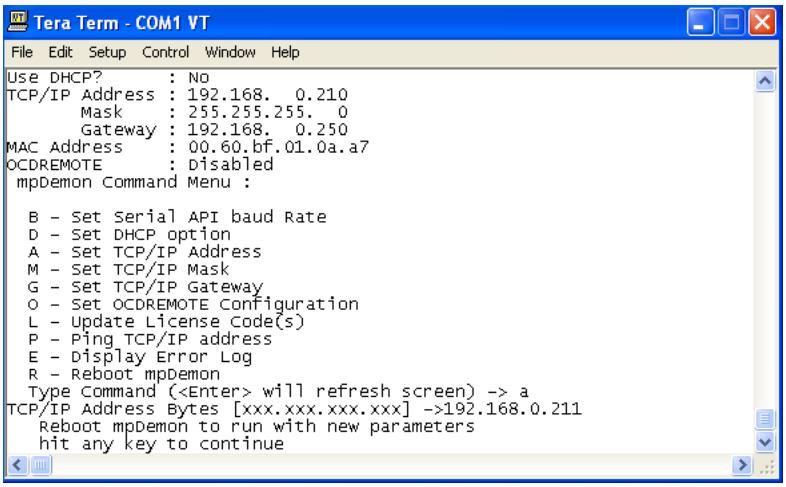

Tera Term - COM1 VT
File Edit Setup Control Window Help
API Baud Rate : 115200
Use DHCP? : No
TCP/IP Address : 192.168. 0.211
Mask : 255.255.255. 0
Gateway : 192.168. 0.250
MAC Address : 00.60.bf.01.0a.a7
OCDREMOTE : Disabled
mpDemon Command Menu :
B - Set Serial API baud Rate
D - Set DHCP option
A - Set TCP/IP Address
M - Set TCP/IP Mask
G - Set TCP/IP Gateway
O - Set OCDREMOTE Configuration
L - Update License Code(s)
P - Ping TCP/IP address
E - Display Error Log
R - Reboot mpDemon
Type Command (<Enter> will refresh screen) -> d
Use DHCP? (Y/N) ->y

Figure 6.58. Enter DHCP Hostname


Tera Term - COM1 VT
File Edit Setup Control Window Help
Macraigor Systems mpDemon(tm) version 3.0.0.0-beta1
API Baud Rate : 115200
Use DHCP? : Yes
DHCP Hostname : MPD000001
TCP/IP Address : 192.168. 0.210
Mask : 255.255.255. 0
Gateway : 192.168. 0.250
MAC Address : 00.60.bf.01.0a.a7
OCDREMOTE : Disabled
mpDemon Command Menu :
B - Set Serial API baud Rate
D - Set DHCP option
N - Set DHCP hostName
O - Set OCDREMOTE Configuration
L - Update License Code(s)
P - Ping TCP/IP address
E - Display Error Log
R - Reboot mpDemon
Type Command (<Enter> will refresh screen) -> n
DHCP hostname ->192.168.0.250

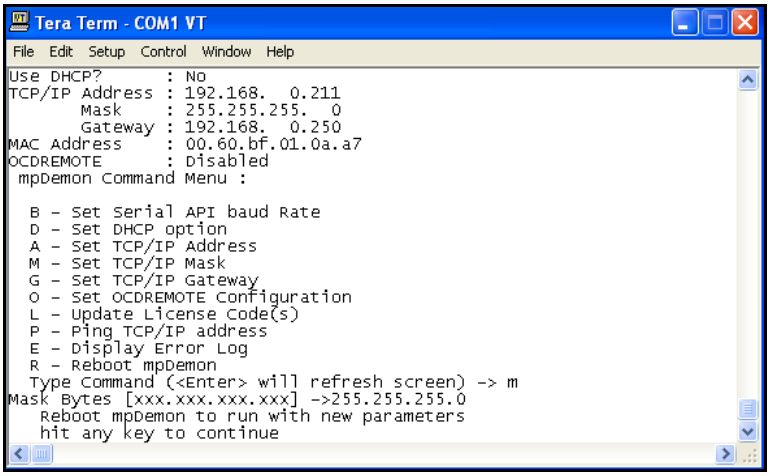
8. Skip this step if using DHCP. If entering your network settings manually instead of using DHCP, then select ‘A’ in the mpDemon command menu followed by Enter – see Figure 6.59.

Figure 6.59 Enter TCP/IP Address


Tera Term - COM1 VT
File Edit Setup Control Window Help
Use DHCP? : No
TCP/IP Address : 192.168. 0.210
Mask : 255.255.255. 0
Gateway : 192.168. 0.250
MAC Address : 00.60.bf.01.0a.a7
OCDREMOTE : Disabled
mpDemon Command Menu :
B - Set Serial API baud Rate
D - Set DHCP option
A - Set TCP/IP Address
M - Set TCP/IP Mask
G - Set TCP/IP Gateway
O - Set OCDREMOTE Configuration
L - Update License Code(s)
P - Ping TCP/IP address
E - Display Error Log
R - Reboot mpDemon
Type Command (<Enter> will refresh screen) -> a
TCP/IP Address Bytes [xxx.xxx.xxx.xxx] ->192.168.0.211
Reboot mpDemon to run with new parameters
hit any key to continue

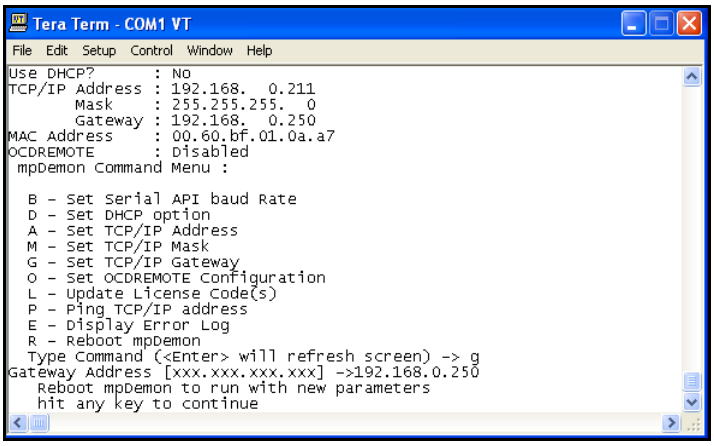
9. Skip this step if using DHCP.

Figure 6.60. Enter Mask


Tera Term - COM1 VT
File Edit Setup Control Window Help
Use DHCP? : No
TCP/IP Address : 192.168. 0.211
Mask : 255.255.255. 0
Gateway : 192.168. 0.250
MAC Address : 00.60.bf.01.0a.a7
OCDREMOTE : Disabled
mpDemon Command Menu :
B - Set Serial API baud Rate
D - Set DHCP option
A - Set TCP/IP Address
M - Set TCP/IP Mask
G - Set TCP/IP Gateway
O - Set OCDREMOTE Configuration
L - Update License Code(s)
P - Ping TCP/IP address
E - Display Error Log
R - Reboot mpDemon
Type Command (<Enter> will refresh screen) -> m
Mask Bytes [xxx.xxx.xxx.xxx] ->255.255.255.0
Reboot mpDemon to run with new parameters
hit any key to continue

10. Skip this step if using DHCP.

Figure 6.61. Enter Gateway Address


Tera Term - COM1 VT
File Edit Setup Control Window Help
Use DHCP? : No
TCP/IP Address : 192.168. 0.211
Mask : 255.255.255. 0
Gateway : 192.168. 0.250
MAC Address : 00.60.bf.01.0a.a7
OCDREMOTE : Disabled
mpDemon Command Menu :
B - Set Serial API baud Rate
D - Set DHCP option
A - Set TCP/IP Address
M - Set TCP/IP Mask
G - Set TCP/IP Gateway
O - Set OCDREMOTE Configuration
L - Update License Code(s)
P - Ping TCP/IP address
E - Display Error Log
R - Reboot mpDemon
Type Command (<Enter> will refresh screen) -> g
Gateway Address [xxx.xxx.xxx.xxx] ->192.168.0.250
Reboot mpDemon to run with new parameters
hit any key to continue

11. Next, enter ‘O’ to configure the OCD Remote settings. These settings include the JTAG speed, number of scan devices, processor device selection, and TCP/IP port number (recommend 8888). Recommend setting the JTAG speed to 4 initially, and experiment with other settings when you make a successful connection (lower the setting, the faster the connection speed) – see Figure 6.62. Since the mpDemon can scan more than one processor device through a single JTAG, you can have multiple CPUs on one board with a single JTAG interface; for a single processor, enter ‘1’ – see Figure 6.63.

Figure 6.62. Select JTAG Speed

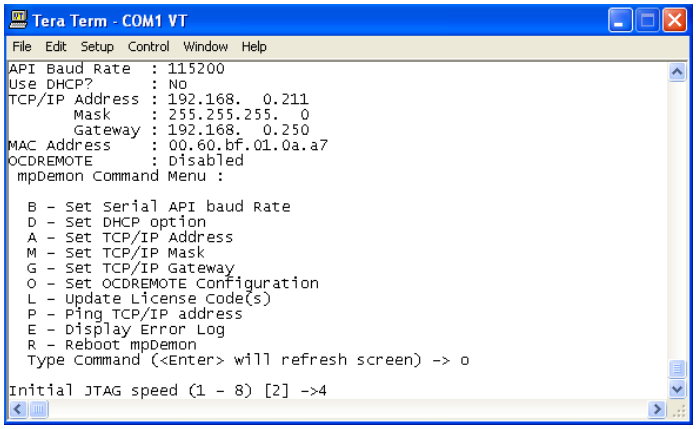
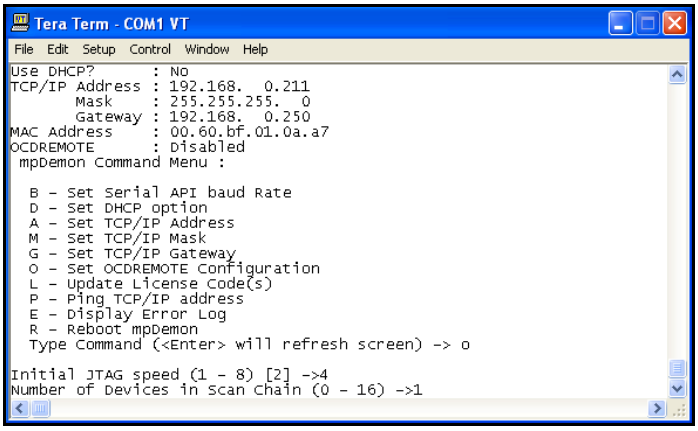

Tera Term - COM1 VT
File Edit Setup Control Window Help
API Baud Rate : 115200
Use DHCP? : No
TCP/IP Address : 192.168. 0.211
Mask : 255.255.255. 0
Gateway : 192.168. 0.250
MAC Address : 00.60.bf.01.0a.a7
OCDREMOTE : Disabled
mpDemon Command Menu :
B - Set Serial API baud Rate
D - Set DHCP option
A - Set TCP/IP Address
M - Set TCP/IP Mask
G - Set TCP/IP Gateway
O - Set OCDREMOTE Configuration
L - Update License Code(s)
P - Ping TCP/IP address
E - Display Error Log
R - Reboot mpDemon
Type Command (<Enter> will refresh screen) -> o
Initial JTAG speed (1 - 8) [2] ->4

Figure 6.63. Select Number of Scan Devices


Tera Term - COM1 VT
File Edit Setup Control Window Help
Use DHCP? : No
TCP/IP Address : 192.168. 0.211
Mask : 255.255.255. 0
Gateway : 192.168. 0.250
MAC Address : 00.60.bf.01.0a.a7
OCDREMOTE : Disabled
mpDemon Command Menu :
B - Set Serial API baud Rate
D - Set DHCP option
A - Set TCP/IP Address
M - Set TCP/IP Mask
G - Set TCP/IP Gateway
O - Set OCDREMOTE Configuration
L - Update License Code(s)
P - Ping TCP/IP address
E - Display Error Log
R - Reboot mpDemon
Type Command (<Enter> will refresh screen) -> o
Initial JTAG speed (1 - 8) [2] ->4
Number of Devices in Scan Chain (0 - 16) ->1

12. Next, select the processor device for your connection. All of the options appear on the screen – see Figure 6.64. The proper Personality Module (connection cable) must be used with the appropriate processor device selected.

13. Enter the TCP/IP port number (recommend 8888) – see Figure 6.65.

14. Now reboot the mpDemon by entering 'R' and pressing the Enter key – see Figure 6.66. The banner screen will show all of your new settings. You have now configured the mpDemon for use and now can perform a configuration test. Go onto the next step.

Figure 6.64. Select Device

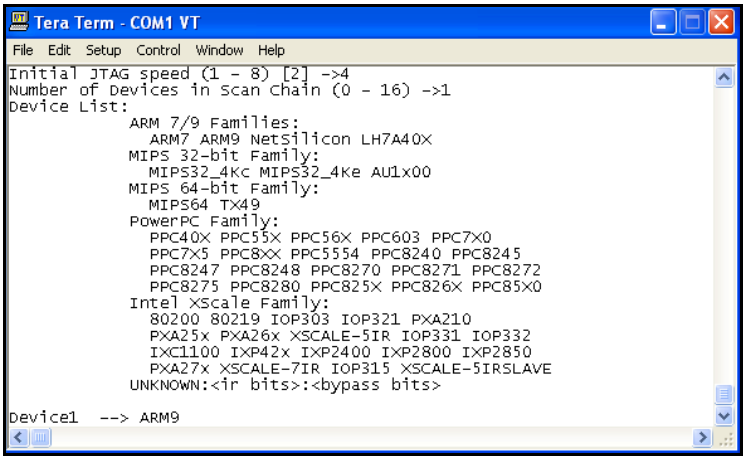


Figure 6.65. Select Port Number

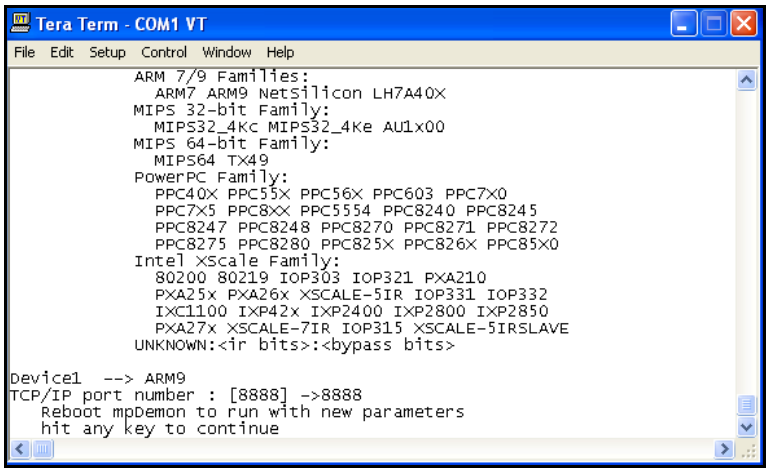
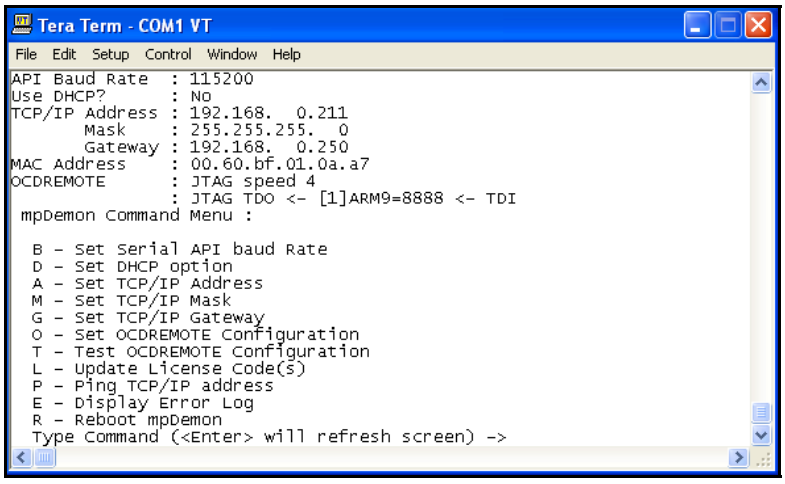


Figure 6.66. Reboot the mpDemon


Tera Term - COM1 VT
File Edit Setup Control Window Help
API Baud Rate : 115200
Use DHCP? : No
TCP/IP Address : 192.168. 0.211
Mask : 255.255.255. 0
Gateway : 192.168. 0.250
MAC Address : 00.60.bf.01.0a.a7
OCDREMOTE : JTAG speed 4
: JTAG TDO <- [1]ARM9=8888 <- TDI
mpDemon Command Menu :
B - Set Serial API baud Rate
D - Set DHCP option
A - Set TCP/IP Address
M - Set TCP/IP Mask
G - Set TCP/IP Gateway
O - Set OCDREMOTE Configuration
T - Test OCDREMOTE Configuration
L - Update License Code(s)
P - Ping TCP/IP address
E - Display Error Log
R - Reboot mpDemon
Type Command (<Enter> will refresh screen) ->
Type Command (<Enter> will refresh screen) ->
Type Command (<Enter> will refresh screen) ->

15. Press ‘T’ and the Enter key to test the OCDREMOTE configuration; you will be prompted as indicated in Figure 6.67 for a JTAG speed. Press ‘1’ and the Enter key to see if the staus returns a correct response. The status should read ‘Stopped’. Figure 6.67 shows ‘Running’; therefore, the JTAG speed is set too fast. Repeat this step except enter a higher number – see Figure 6.68, which shows the correct status, ‘Stopped’.

Figure 6.67. mpDemon Configuration Test

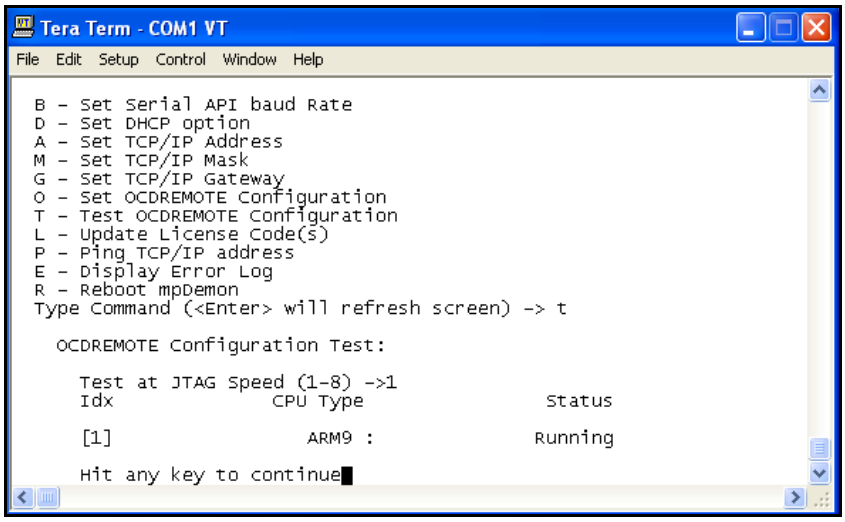
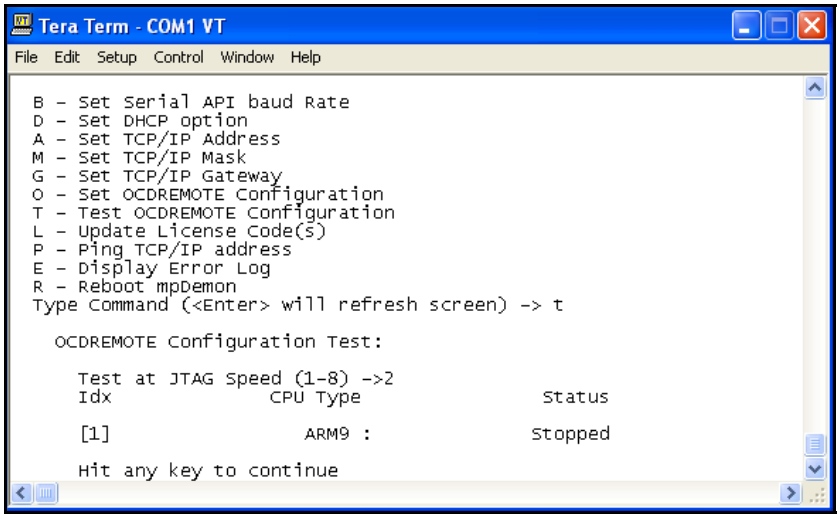

Tera Term - COM1 VT
File Edit Setup Control Window Help
B - Set Serial API baud Rate
D - Set DHCP option
A - Set TCP/IP Address
M - Set TCP/IP Mask
G - Set TCP/IP Gateway
O - Set OCDREMOTE Configuration
T - Test OCDREMOTE Configuration
L - Update License Code(s)
P - Ping TCP/IP address
E - Display Error Log
R - Reboot mpDemon
Type Command (<Enter> will refresh screen) -> t
OCDREMOTE Configuration Test:
Test at JTAG Speed (1-8) ->1
Idx CPU Type Status
[1] ARM9 : Running
Hit any key to continue

Figure 6.68. mpDemon Configuration Test


Tera Term - COM1 VT
File Edit Setup Control Window Help
B - Set Serial API baud Rate
D - Set DHCP option
A - Set TCP/IP Address
M - Set TCP/IP Mask
G - Set TCP/IP Gateway
O - Set OCDREMOTE Configuration
T - Test OCDREMOTE Configuration
L - Update License Code(s)
P - Ping TCP/IP address
E - Display Error Log
R - Reboot mpDemon
Type Command (<Enter> will refresh screen) -> t
OCDREMOTE Configuration Test:
Test at JTAG Speed (1-8) ->2
Idx CPU Type Status
[1] ARM9 : Stopped
Hit any key to continue

Conclusion of the mpDemon Setup

The configuration of the mpDemon only has to be entered once for a given target system. It is stored in the mpDemon's flash and will remain active until it is changed by the user. To use the web-based interface to configure the mpDemon, you can plug in the Ethernet cable to your network and open a web browser and enter the TCP/IP address that you configured into the mpDemon (e.g., http://192.168.0.220). Changes to a configuration can then be made through the browser instead of the serial connection.

Visual GDB Startup with the mpDemon

1. Start the Visual GDB debugger with a binary with debug symbols, <filename>, and click on 'File|Target Setting' and following the procedures below to completely setup communications within Visual GDB:

   $ <target-alias>-gdbtk <filename> (enter)
   (you must be in the directory where the binary, <filename>, is located; if using Visual X-Tools to start the Visual GDB debugger, then simply click on the icon or menu item under Build)

2. Click on 'Target' combo-box and select 'Remote/TCP'.

3. Enter the 'TCP/IP address' of the mpDemon.

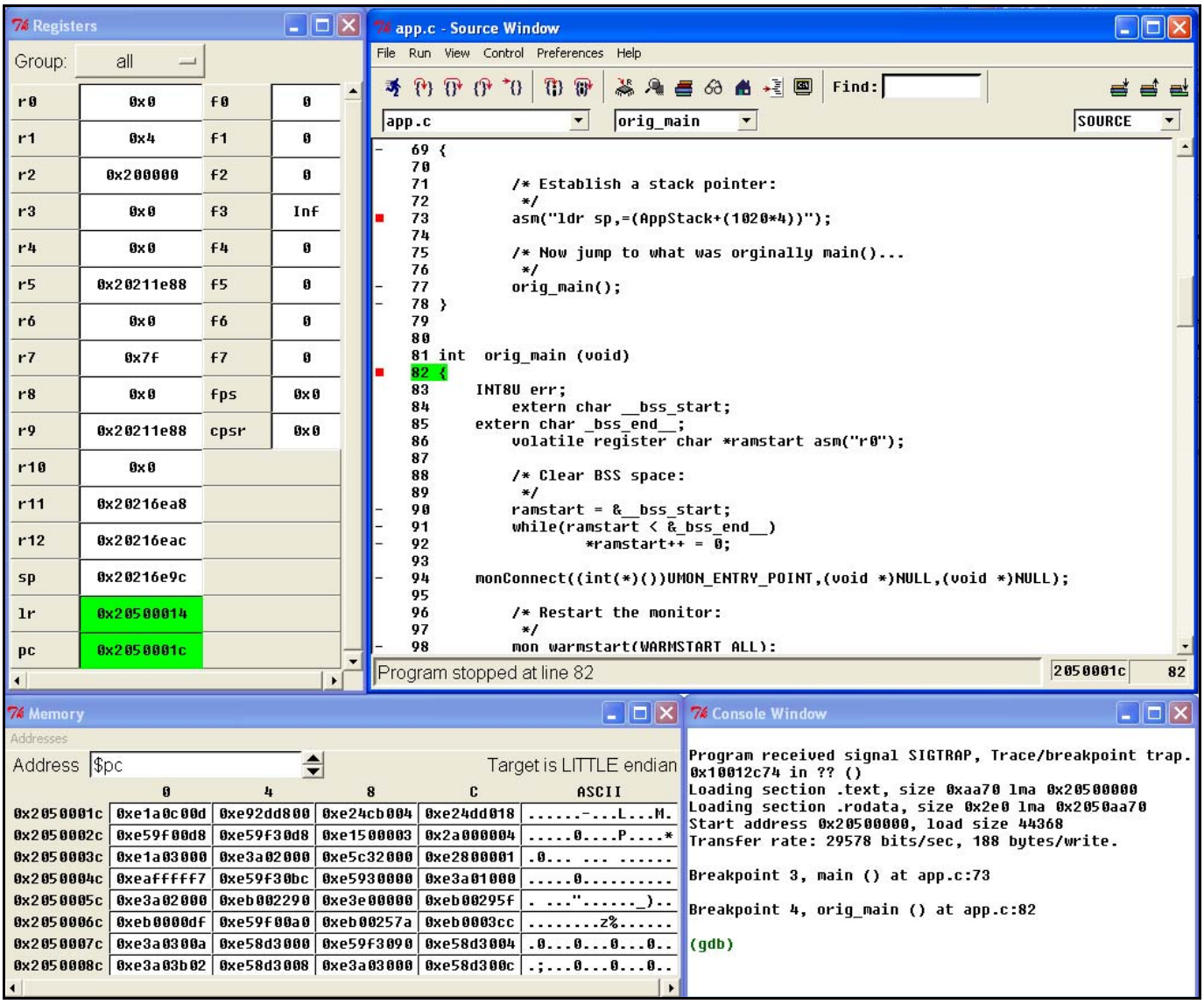
4. Enter 'port address'— if setup as instructed, it should be '8888'.

5. Select the desired check boxes.

6. Click on 'More Options' enter any desired 'Run Options' and click 'OK'. If debugging on the hardware target, select the run option 'continue' since a program cannot be run in the traditional sense through the JTAG.

7. Set your breakpoints, if necessary, and click on 'run', and thereafter you can single step or continue. Click on 'View' to select view options like registers, memory, console, function browser, etc.. Review Section 5 for more details on how to use Visual GDB. Figure 6.69 shows a screenshot of a program running and stopping at its first breakpoint; it also shows views of registers, memory, and console along with the main source window.

Figure 6.69. Visual GDB Screenshot



Section 7. Introduction to Cygwin

7.1 Introducing Cygwin

One of the largest problems developers face today is supporting their applications on disparate platforms. Many of the new Windows workstations are being added to an environment already populated by Solaris, HP/UX, AIX and Linux systems.

Using Cygwin, developers can manage heterogeneous environments in a consistent, efficient way. Cygwin brings a standard UNIX/Linux shell environment, including many of its most useful commands, to the Windows platform so software managers can effectively deploy trained staff, and leverage existing investments in UNIX/Linux source code and shell scripts.

Cygwin delivers the Free Source standard GNU GCC compiler and GDB debugger on Windows. In addition, it provides for a standard UNIX/Linux development environment on Windows including APIs and command shells. The 'Cygwin.dll' library, included with Cygwin, delivers the interesting subset of UNIX SVR4, BSD, and POSIX APIs to enable quick ports of UNIX/Linux applications to the Windows platform. The license for 'Cygwin.dll' is covered by the GNU General Public License and only Open/Free Source projects can be built and distributed with the 'Cygwin.dll'. The commercial software with the 'Cygwin.dll' may be purchased separately. The Microcross i.MX GNU X-Tools are bound to the GPL license; however, users of i.MX GNU X-Tools who use the preconfigured Newlib C and Math libraries are not subject to the GPL license. The Newlib is completely unrestricted and may be used with confidence that your proprietary source code is protected.

In addition to providing an environment for Open/Free Source development on the Windows platform, Cygwin delivers a large collection of these Open/Free Source programs in a *contributed* directory. Through this collection of Open/Free Source UNIX/Linux programs, Cygwin provides basic commands and utilities for remote system administration, including all the standard remote internal daemons (e.g., ftp, telnet, tar, cpio, rlogin.)

7.2 Cygwin Key Features

- Provides a shell environment for the Windows Platform
- Comes with the latest Intel Pentium II optimizing GNU native compiler
- Includes the largest collection of Open Source Linux projects ported to Windows, making it the standard Open Source distribution for the Windows platform
- The Cygwin.dll portability library provides the capability to port UNIX / Linux Open Source projects to Windows
- Cygwin.dll allows for compiling UNIX applications on Windows.

7.3 Cygwin Components

- UNIX / Linux shell environment (such as: bash)
- Open Source Internet daemons (such as: telnet)
- GNU Development Tools optimized for the Intel Pentium family of processors
- 'Cygwin.dll' portability library

7.3.1 Cygwin Package List – tools, utilities, and programs that make up Cygwin

bash	bc	binutils	bison
byacc	bzip2	cpio	cvs
cygwin	diffutils	expect	file
fileutils	findutils	flex	fortune
ftp	gas	gawk	gcc
GDB	gnuchess	gprof	grep

groff	Gzip	inetutils	itcl
itk	ksh	ld	less
libcurses	libjpeg	libtermcap	libtiff
login	m4	make	man
more	Ncurses	perl	rx/regex
sed	shell-utils	tar	tcl
tcsh	telnet	termcap	texinfo
textutuils	time	tix	tk
vim / vi	wget	which	Xemacs
zip/unzip	zlib		

7.3.2 User Information – How to use the tools, utilities, and programs in Cygwin

Start a i.MX GNU X-Tools shell and type ‘man‘ and the name of the tool, utility, or program you wish to learn about at the command prompt. Type ‘q’ to quit ‘man’ pages. The Cygwin CD-ROM provided with the Cygwin version of i.MX GNU X-Tools has an option to install Cygwin documentation. We highly recommend that you install the documentation and take advantage of some very powerful programs, tools, and utilities. The documentation that you have the option of installing is html based and you can find the documentation in a directory named html in the Cygwin root directory. Create a desktop shortcut if desire to reach the help index in the html subdirectory of Cygwin. Below is a short description of the tools, utilities, and programs listed above. The best place to get more information on Cygwin is to go to the Cygwin home page on the web: www.cygwin.com

Tool/Utility/Program	Description
bash	GNU Bourne-Again Shell; sh-compatible command language interpreter; IEEE POSIX shell
bc	An arbitrary precision calculator language
binutils	An assortment of GNU utilities including ar, nm, objcopy, ranlib, size, strings, strip, c++filt, addr2line, nlmconv, windres
bison	GNU Project parser generator (yacc replacement)
byacc	An LALR(1) parser generator
bzip2	A compression utility
cpio	Copy files to and from archives (like tar)
cvs	Concurrent Versions System; a version control system for managing source code
Cygwin	A Unix / Linux portability / application layer on Windows OS
diffutils	Includes file comparison utilities: cmp, diff, diff3, sdiff
expect	Programmed dialog with interactive programs
file	Determine file type
fileutils	GNU file utilities including: chmod, chown/chgrp, cp, dd, df, dir, dircolors, du, install/ginstall, ls , ln, mkdir, mkfifo/mknod, mv, rm, rmdir, sync, touch, vdir
findutils	Programs for finding files
flex	Replacement for the lex scanner generator
fortune	Print a random, hopefully interesting, adage
ftp	ARPANET file transfer program
gas	GNU assembler
gawk	A pattern-scanning and text-processing program
gcc	The GNU Compiler Collection with C compiler
GDB	The GNU Debugger
gnuchess	GNU’s Chess game – play against a user or it plays against itself
gprof	Display call graph profile data
grep	Print lines matching a pattern – a very powerful search utility
groff	Front end for the gruff document formatting system
gzip	Compress or expand files
inetutils	All Internet utilities in Cygwin
itcl	Object oriented extensions to Tcl
itk	Framework for building mega-widgets in Tcl/Tk
ksh	Public domain Korn shell

Tool/Utility/Program	Description
ld	GNU linker
less	Opposite of more; less does not have to read the entire input file before starting
libcurses	Library containing all drivers and files for terminal server
libjpeg	Library for reading and writing JPEG files
libtiff	Library for reading and writing TIFF files
M4	Implementation of a traditional Unix macro processor
make	GNU make supports POSIX 1003.2 and is used to manage large programming projects
man	Format and display on-line manual pages
more	Display paginator; supports backward and forward movement though a file and searches
ncurses	CRT screen handling and optimization package – used as: #include <curses.h> library routines giving the user a terminal-independent method of updating character screens with reasonable optimization.
perl	Practical Extraction and Report Language; designed for scanning files, generating reports, and manipulating text, and other small programming tasks; combines many features of C, awk, sed, and shell programming
sed	A stream editor based on the line editor ed
shell-utils	GNU Shell Utilities designed for use in shell scripts and batch files
tar	Program used to create and manipulate Tape ARchives (tar).
tcl	A basic programming language very useful for graphical interface facilities;
telnet	User interface to the TELNET protocol, which is used to communicate with another host computer.
texinfo	Set of utilities that generates both printed manuals and on-line hepertext-style documentation
textutils	GNU Text Utilities to manipulate textual data
tix	Manipulate Tix internal state; options are set using the X resource database
tk	A toolkit of widgets, graphical objects, for the Tcl programming system
Vim renamed to vi	Vi Improved, a programmer's text editor
which	Show full path of commands
Xemacs	Emacs: the Next Generation of text editor; a very powerful source code editor
zip/unzip	A compression utility; compress and uncompress files

Appendix 1. ARM Toolsuite

MODEL (target alias):	ARM -- Advanced RISC Machines (alias: arm-elf)
MFR:	Freescale Semiconductor, Inc.
CPU TYPE:	RISC Von Neumann, 32-Bit Fixed Instructions, 16-Bit Thumb mode
GP REGS/SIZE:	16 / 32-Bit
ADDR SPACE:	64MB Linear -- Endian: Selectable
CYCLE RATE:	Core Dependent
APPROX INSTRUCTION THROUGHPUT (PIPELINED):	Depends on core variant
# INSTRUCTIONS:	11 TYPES
# ADDR MODES:	5 Load/Store
FP INSTRUCTIONS:	None
INTERRUPTS:	8 Exceptions, 2 External Interrupts (normal, fast)
VARIANTS:	ARM 2/250/3/6/60/600/610/620
	ARM 7/7m/7d/7dm/7di/7dmi/70/700/700i/710/71xxc/75xxfe/7tdmi
	ARM 8/810/9/9e/920/920t,926ejs,940t,9tdmi,
	ARM 1020t/1026ejs, StrongARM 110/1100
	ARM 1136js/1136jfs/XScale/iwmmxt/ep9312
	Architectures: armv2, armv2a, armv3, armv3m, armv4, armv4t
	armv5, armv5t, armv5te, armv6j, iwmmxt
ATTRIBUTES:	Based on current independent studies, the ARM is the most popular RISC processor for embedded applications. Its architectural specification is highly scalable, and a number of variants have been defined and produced by several manufacturers. As a result, a wide range of configurations and performance levels exist, giving embedded designers many flexible design options. The simple instruction set architecture is easy to learn, yet powerful and optimized for compilers.
REFERENCES:	www.arm.com

The ARM Toolsuite consists of the following tools:

Tool Name	Tool Description	Version
arm-elf-gcc	GNU Compiler Collection (GCC)	3.40
arm-elf-g++	C++ compiler	3.40
arm-elf-as arm-elf-ld arm-elf-addr2line arm-elf-ar arm-elf-gcov arm-elf-nm arm-elf-objcopy arm-elf-objdump arm-elf-ranlib arm-elf-readelf arm-elf-size arm-elf-strings Arm-elf-strip	GNU assembler (as) GNU linker (ld) Converts addresses to file names & line # Creates object code archives Test coverage program for analysis Lists symbols from object files Copies and translates object files Displays information from object files Generates index to archive contents Displays information about ELF objects Lists file section sizes and total sizes Lists printable strings from files Strips debug symbols from binaries	2.15
arm-elf-gdb arm-elf-gdbtk arm-elf-run	GDB debugger Visual GDB Instruction Set Simulator	6.1
libc.a and libm.a	Unrestricted Newlib C and Math Libraries	1.12
libstdc++.a	GNU Standard C++ and Template Library	3.40

File Count in Distribution

Type	Path	Files: Win32	Files: Linux
binary images	/usr/bin/arm-elf-*	22	25
binary images	/usr/arm-elf/*	10	10
library files	/usr/arm-elf/lib/*	281	281
include files	/usr/arm-elf/include/*	306	309
compiler/lib	/usr/lib/gcc-lib/arm-elf/*	128	128
man pages	/usr/man/man1/arm-elf-*	22	23
	Total Files:	769	776

TARGET CPU DEPENDENT INFORMATION

This toolsuite comprises a complete development environment for the ARM processor family. Included in this section of the Appendix are the required compiler and linker flags, a list of debugger targets for both command line GDB and Visual GDB, machine compiler and assembler dependent options, and notes.

Required Compiler / Linker Flags

The following compiler/linker options are required to build for execution with the simulator.

CFLAGS = (none required)
LFLAGS = (none required)

Command Line GDB Debugger Targets

- async -– Use a remote computer via a serial line.
- cisco -– Use a remote machine via TCP.
- exec -– Use an executable file as a target.
- extended-async -– Use a remote computer via a serial line.
- extended-remote -– Use a remote computer via a serial line.
- rdi -– Connects to target running ARM Angel Monitor. To connect to an external target using the rdi (Angel) monitor, use the GDB command ‘target rdi N’ where N equals ‘1’ or ‘2’ to select serial port 1 or 2 as the interface port.

- rdp — Use a remote ARM system that uses the ARM Remote Debugging Protocol.
- remote -- Allows the GDB to connect to a remote target running the GDB remote debugging protocol via a serial port or a TCP socket connection. The GDB debugger can connect to a board via Abatron's BDI2000 or EPI's JEENI JTAG debug agents across Ethernet. Any debug agent that communicates using the GDB Stub protocol can be accessed through GDB/Visual GDB.
- sim -- The GDB debugger contains a integrated simulator based target that can load and execute compiled programs.

Visual GDB Debugger Targets

- Cisco/Serial
- Cisco/TCP
- ARM Angel/Serial
- ARM Angel/Ethernet
- ARM Remote/Serial
- ARM Remote/TCP
- Remote/Serial
- Remote/TCP
- Simulator
- GDBserver/Serial
- GDBserver/TCP

Compiler Machine Dependent Options

Section 3 describes all of the common compiler, linker, and assembler options used in Microcross GNU cross-tools; in addition, these `-m' options are defined for ARM architectures:

Option	Description
-mapcs-frame	Generate a stack frame that is compliant with the ARM Procedure Call Standard for all functions, even if this is not strictly necessary for correct execution of the code. Specifying '-fomit-frame-pointer' with this option will cause the stack frames not to be generated for leaf functions. The default is '-mno-apcs-frame'.
-mapcs	This is a synonym for '-mapcs-frame'.
-mapcs-32	Generate code for a processor running with a 32-bit program counter, and conforming to the function calling standards for the APCS 32-bit option. This option replaces the '-m6' option of previous releases of the compiler and is now the default.
-mapcs-stack-check	Generate code to check the amount of stack space available upon entry to every function (that actually uses some stack space). If there is insufficient space available then either the function '__rt_stkovf_split_small' or '__rt_stkovf_split_big' will be called, depending upon the amount of stack space required. The run time system is required to provide these functions. The default is '-mno-apcs-stack-check', since this produces smaller code.
-mthumb-interwork	Generate code which supports calling between the ARM and Thumb instruction sets. Without this option the two instruction sets cannot be reliably used inside one program. The default is '-mno-thumb-interwork', since slightly larger code is generated when '-mthumb-interwork' is specified.
-mno-sched-prolog	Prevent the reordering of instructions in the function prolog, or the merging of those instruction with the instructions in the function's body. This means that all functions will start with a recognizable set of instructions (or in fact one of a choice from a small set of different function prologues), and this information can be used to locate the start if functions inside an executable piece of code. The default is '-msched-prolog'.
-mhard-float	Generate output containing floating-point instructions. Note: The

Option	Description
	Instruction Set Simulator (ISS) will only run code generated with the default software floating point emulation.
-msoft-float	This is the default, and it is not necessary to specify it. The Instruction Set Simulator (ISS) will only run code generated with the default software floating point emulation.
-mlittle-endian	Generate code for a processor running in little-endian mode. This is the default for all standard configurations.
-mbig-endian	Generate code for a processor running in big-endian mode; the default is to compile code for a little-endian processor.
-mwords-little-endian	This option only applies when generating code for big-endian processors. Generate code for a little-endian word order but a big-endian byte order. That is, a byte order of the form '32107654'. Note: this option should only be used if you require compatibility with code for big-endian ARM processors generated by versions of the compiler prior to 2.8.
-malignment-traps	Generate code that will not trap if the MMU has alignment traps enabled. On ARM architectures prior to ARMv4, there were no instructions to access half-word objects stored in memory. However, when reading from memory a feature of the ARM architecture allows a word load to be used, even if the address is unaligned, and the processor core will rotate the data as it is being loaded. This option tells the compiler that such misaligned accesses will cause a MMU trap and that it should instead synthesize the access as a series of byte accesses. The compiler can still use word accesses to load half-word data if it knows that the address is aligned to a word boundary. This option is ignored when compiling for ARM architecture 4 or later, since these processors have instructions to directly access half-word objects in memory.
-mno-alignment-traps	Generate code that assumes that the MMU will not trap unaligned accesses. This produces better code when the target instruction set does not have half-word memory operations (implementations prior to ARMv4). Note that you cannot use this option to access unaligned word objects, since the processor will only fetch one 32-bit aligned object from memory. The default setting for most targets is '-mno-alignment-traps', since this produces better code when there are no half-word memory instructions available.
-mcpu=<name>	This specifies the name of the target ARM processor. GCC uses this name to determine what kind of instructions it can use when generating assembly code. Permissible names are: arm2, arm250, arm3, arm6, arm60, arm600, arm610, arm620, arm7, arm7m (the default setting), arm7d, arm7dm, arm7di, arm7dmi, arm70, arm700, arm700i, arm710, arm710c, arm7100, arm7500, arm7500fe, arm7tdmi, arm8, strongarm, strongarm110, strongarm1100, arm8, arm810, arm9, arm920, arm920t, arm9tdmi, arm9e, arm926ejs, arm1020t, arm1026ejs, arm1136js, arm1136jfs, iwmmxt, and ep9312.
-mtune=<name>	This option is very similar to the '-mcpu=' option, except that instead of specifying the actual target processor type, and hence restricting which instructions can be used, it specifies that GCC should tune the performance of the code as if the target were of the type specified in this option, but still choosing the instructions that it will generate based on the CPU specified by a '-mcpu=' option. For some arm implementations better performance can be obtained by using this option.
-march=<name>	This specifies the name of the target ARM architecture. GCC uses this name to determine what kind of instructions it can use when generating assembly code. This option can be used in conjunction with or instead of the '-mcpu=' option. Permissible names are: armv2, armv2a, armv3, armv3m, armv4 (the default setting), armv4t, armv5, armv5te, arm6j, iwmmxt.

Option	Description
-mfpe=<number> -mfp=<number>	This specifies the version of the floating-point emulation available on the target. Permissible values are 2 and 3. '-mfp=' is a synonym for '-mfpe=' to support older versions of GCC.
-mstructure-size-boundary=<n>	The size of all structures and unions will be rounded up to a multiple of the number of bits set by this option. Permissible values are 8 and 32. The default value varies for different toolsuites. For the COFF targeted toolsuite the default value is 8. Specifying the larger number can produce faster, more efficient code, but can also increase the size of the program. The two values are potentially incompatible. Code compiled with one value cannot necessarily expect to work with code or libraries compiled with the other value, if they exchange information using structures or unions. Programmers are encouraged to use the 32 value as future versions of the toolsuite may default to this value.
-mabort-on-noreturn	Generate a call to the function abort at the end of a noreturn function. It will be executed if the function tries to return.
-mlong-calls -mno-long-calls	Tells the compiler to perform function calls by first loading the address of the function into a register and then performing a subroutine call on this register. This switch is needed if the target function will lie outside of the 64 megabyte addressing range of the offset based version of subroutine call instruction. Even if this switch is enabled, not all function calls will be turned into long calls. The heuristic is that static functions, functions which have the 'short-call' attribute, functions that are inside the scope of a '#pragma no_long_calls' directive and functions whose definitions have already been compiled within the current compilation unit, will not be turned into long calls. The exception to this rule is that weak function definitions, functions with the `long-call' attribute or the `section' attribute, and functions that are within the scope of a '#pragma long_calls' directive, will always be turned into long calls. This feature is not enabled by default. Specifying '--no-long-calls' will restore the default behavior, as will placing the function calls within the scope of a '#pragma long_calls_off' directive. Note these switches have no effect on how the compiler generates code to handle function calls via function pointers.
-msingle-pic-base	Treat the register used for PIC addressing as read-only, rather than loading it in the prologue for each function. The run-time system is responsible for initializing this register with an appropriate value before execution begins.
-mpic-register=<reg>	Specify the register to be used for PIC addressing. The default is r10 unless stack checking is enabled, when r9 is used. Must be used with –fpic option to be valid.
-mcirrus-fix-invalid-insns	Insert NOPs into the instruction stream to in order to work around problems with invalid Maverick instruction combinations. This option is only valid if the -mcpu=ep9312 option has been used to enable generation of instructions for the Cirrus Maverick floating point co-processor. This option is not enabled by default, since the problem is only present in older Maverick implementations. The default can be re-enabled by use of the -mno-cirrus-fix-invalid-insns switch.
-mpoke-function-name	Write the name of each function into the text section, directly preceding the function prologue. The generated code is similar to this: t0 .ascii "arm_poke_function_name", 0 .align t1

Option	Description

	.word 0xff000000 + (t1 - t0)
	arm_poke_function_name
	mov ip, sp
	stmfd sp!, {fp, ip, lr, pc}
	sub fp, ip, #4
	
	When performing a stack backtrace, code can inspect the value of pc stored at fp + 0. If the trace function then looks at location pc - 12 and the top 8 bits are set, then we know that there is a function name embedded immediately preceding this location and has length ((pc[-3]) & 0xff000000).

-mthumb	Generate code for the 16-bit Thumb instruction set. The default is to use the 32-bit ARM instruction set. Generally this option is used with the –mthumb-interwork.
-mtpcs-frame	Generate a stack frame that is compliant with the Thumb Procedure Call Standard for all non-leaf functions. (A leaf function is one that does not call any other functions.) The default is -mno-tpcs-frame.
-mtpcs-leaf-frame	Generate a stack frame that is compliant with the Thumb Procedure Call Standard for all leaf functions. (A leaf function is one that does not call any other functions.) The default is -mno-apcs-leaf-frame.
-mcallee-super-interworking	Gives all externally visible functions in the file being compiled an ARM instruction set header which switches to Thumb mode before executing the rest of the function. This allows these functions to be called from non-interworking code.
-mcaller-super-interworking	Allows calls via function pointers (including virtual functions) to execute correctly regardless of whether the target code has been compiled for interworking or not. There is a small overhead in the cost of executing a function pointer if this option is enabled.

Compiler-Defined Symbols and Standard Search Directories
From the command line in i.MX GNU X-Tools Shell, type:

```
$ xtools arm-elf  (enter)
arm-elf$ gcc –v –E –  (enter)
```

You will get all of the compiler-defined symbols printed out on the screen. Press Ctrl-C to exit this mode.

Assembler Machine Dependent Options
The following options are available when as is configured for the ARM processor family.

Options	Description
-marm2	Arm 2 processor
-marm250 | -marm3	Arm 250 and Arm 3 processor
-marm6	Arm 6 processor
-marm7[tdmi]	Arm 7 processors
-marm8[10]	Arm 8 processors
-marm9[20][tdmi]	Arm 9 processors
-mstrongarm110[0]]	StrongARM processors
-marmv2 | -marmv2a -marmv3 | -marmv3m -marmv4 | -marmv4t -marmv5 | -marmv5te	Arm architectures

Options	Description
-marmv6 | -marmv6j -mthumb -mall	 Thumb mode Assemble All ARM/Thumb instructions
-mfpe-old -mno-fpu	No float load/store multiples Disable all floating point
-mfpa10 | -mfpa11	FPA10 and 11 co-processor
 -mapcs-32 -mapcs-26 -mthumb-interwork	Select which procedure calling convention is in use: 32-bit APCS 16-bit APCS Code supports Arm/Thumb Interworking
-EB | -EL	Select either big-endian (-EB) or little-endian (-EL) output.

GNU ARM Assembler Quick Reference

A summary of useful commands and expressions for the ARM architecture using the GNU assembler is presented briefly in the concluding portion of this Appendix. Each assembly line has the following format:

[<label>:] [<instruction or directive>} @ comment

Unlike the ARM assembler, using the GNU assembler does not require you to indent instructions and directives. Labels are recognized by the following colon instead of their position at the start of a line. An example follows showing a simple assembly program defining a function 'add' that returns the sum of two input arguments:

```
.section .text, "x"

.global  add                    @ give the symbol add external linkage

add:
        ADD     r0, r0, r1              @ add input arguments
        MOV     pc, lr          @ return from subroutine

                                        @ end of program
```

GNU Assembler Directives for ARM

The follow is an alphabetical listing of the more command GNU assembler directives.

GNU Assembler Directive	Description
.ascii "<string>"	Inserts the string as data into the assembly (like DCB in armasm).
.asciz "<string>"	Like .ascii, but follows the string with a zero byte.
.balign <power_of_2> {,<fill_value> {,<max_padding>} }	Aligns the address to <power_of_2> bytes. The assembler aligns by adding bytes of value <fill_value> or a suitable default. The alignment will not occur if more than <max_padding> fill bytes are required (similar to ALIGN in armasm).
.byte <byte1> {,<byte2>} …	Inserts a list of byte values as data into the assembly (like DCB in armasm).
.code <number_of_bits>	Sets the instruction width in bits. Use 16 for Thumb and 32 for ARM assembly (similar to CODE16 and CODE32 in armasm).
.else	Use with .if and .endif (similar to ELSE in armasm).
.end	Marks the end of the assembly file (usually omitted).
.endif	Ends a conditional compilation code block – see .if, .ifdef, .ifndef (similar to ENDIF in armasm).

GNU Assembler Directive	Description
.endm	Ends a macro definition – see .macro (similar to MEND in armasm).
.endr	Ends a repeat loop – see .rept and .irp (similar to WEND in armasm).
.equ <symbol name>, <value>	This directive sets the value of a symbol (similar to EQU in armasm)
.err	Causes assembly to halt with an error.
.exitm	Exit a macro partway through – see .macro (similar to MEXIT in armasm)
.global <symbol>	This directive gives the symbol external linkage (similar to EXPORT in armasm).
.hword <short1> {,<short2>} …	Inserts a list of 16-bit values as data into the assembly (similar to DCW in armasm).
.if <logical_expression>	Makes a block of code conditional. End the block using .endif (similar to IF in armasm). See also .else.
.ifdef <symbol>	Include a block of code if <symbol> is defined. End the block with .endif.
.ifndef <symbol>	Include a block of code if <symbol> is not defined. End the block with .endif.
.include "<filename>"	Includes the indicated source file (similar to INCLUDE in armasm or #include in C).
.irp <param> {,<val_1>} {,<val_2>} …	Repeats a block of code, once for each value in the value list. Mark the end of the block using a .endr directive. In the repeated code block, use \<param> to substitute the associated value in the value list.

.macro <name> {<arg_1} {,<arg_2>} … {,<arg_N>}	Defines an assembler macro called <name> with N parameters. The macro definition must end with .endm. To escape from the macro at an earlier point, use .exitm. These directives are similar to MACRO, MEND, and MEXIT in armasm. You must precede the dummy macro parameters by \. For example:
	
	.macro SHIFTLEFT a, b
	.if \b < 0
	MOV \a, \a, ASR #-\b
	.exitm
	.endif
	MOV \a, \a, LSL #\b
	.endm

.rept <number_of_times>	Repeats a block of code the given number of times. End with .endr.
<register_name> .req <register_name>	This directive names a register. It is similar to the RN directive in armasm except that you must supply a name rather than a number on the right (e.g., acc .req r0).
.section <section_name> {,"<flags>"}	Starts a new code or data section. Sections in GNU are called .text, a code section, .data, an initialized data section, and .bss, an uninitialized data section. These sections have default flags, and the linker understands the default names (similar directive to the armasm directive AREA). The following are allowable .section flags for ELF format files: <Flag> Meaning a allowable section w writable section x executable section
.set <variable_name>, <variable_value>	This directive sets the value of a variable. It is similar to SETA in armasm.
.space <number_of_bytes> {,<fill_byte>}	Reserves the given number of bytes. The bytes are filled with zero or <fill_byte> if specified (similar to SPACE in armasm).
.word <word1> {,<word2>} …	Inserts a list of 32-bit word values as data into the assembly (similar to DCD in armasm).

Assembler Special Characters / Syntax

Inline comment char:	'@'
Line comment char:	'#'
Statement separator:	';'
Immediate operand prefix:	'#' or '$'

Register Names

General registers:	%r0 - %r15	($0 = const 0)
FP registers:	%f0 - %f7	
Non-saved (temp) regs:	%r0 - %r3, %r12	
Saved registers:	%r4 - %r10	
Stack ptr register:	%sp	
Frame ptr register:	%fp	
Link (retn) register:	%lr	
Program counter:	%ip	
Status register:	$psw	
Status register flags:	xPSR	
(x = C current)	xPSR_all	
(x = S saved)	xPSR_f	
	xPSR_x	
	xPSR_ctl	
	xPSR_fs	
	xPSR_fx	
	xPSR_fc	
	xPSR_cs	
	xPSR_cf	
	xPSR_cx	
	.. and so on	

Arm Procedure Call Standard (APCS) Conventions

Argument registers:	%a0 - %a4	(aliased to %r0 - %r4)
Returned value regs:	%v1 - %v6	(aliased to %r4 - %r9)

Addressing Modes

'rn' in the following refers to any of the numbered registers, but not the control registers.

addr	Absolute addressing mode
%rn	Register direct
[%rn]	Register indirect or indexed
[%rn,#n]	Register based with offset
#imm	Immediate data

Machine Dependent Directives

.arm	Assemble using arm mode
.thumb	Assemble using thumb mode
.code16	Assemble using thumb mode
.code32	Assemble using arm mode
.force_thumb	Force thumb mode (even if not supported)
.thumb_func	Mark entry point as thumb coded (force bx entry)
.ltorg	Start a new literal pool

Opcodes
For detailed information on the machine instruction set, see this manual:

ARM Architecture Reference Manual, Addison-Wesley ISBN 0-201-73719-1

Here is a recommended book to get a lot of system developer information on the ARM architecture.

ARM System Developer's Guide, Morgan Kaufmann Publishers ISBN 1-55860-874-5 (alk.paper), authors: Andrew N. Sloss, Dominic Symes, Chris Wright, 2004

NOTES & ERRATA

Bibliography

1. *Comparing & Merging Files*, David Mackenzie, Paul Eggert, & Richard Stallman, Free Software Foundation, 1996 *

2. *Cygwin For Windows NT*, Cygnus Solutions, Inc., 1999

3. *GNUPro Auxiliary Development Tools*, Red Hat, Inc., 2001

4. *GNUPro Development Tools*, Red Hat, Inc., 2001

5. *GNUPro Compiler Tools*, Cygnus Solutions, Inc., 1998

6. *GNUPro Compiler Tools*, Red Hat, Inc., 2001

7. *GNUPro Debugging Tools*, Cygnus Solutions, Inc., 1998

8. *GNUPro Debugging Tools*, Red Hat, Inc., 2001

9. *GNUPro Libraries*, Cygnus Solutions, Inc., 1998

10. *GNUPro Libraries*, Red Hat, Inc., 2001

11. *GNUPro Utilities*, Cygnus Solutions, Inc., 1998

12. *GNUPro Utilities*, Cygnus Solutions, Inc., 1999

13. *GNUPro Toolkit*, Cygnus Solutions, Inc., 1998

14. *GNUPro Toolkit Getting Started Guide*, Red Hat, Inc., 2001

15. *GNUPro Tools for Embedded Systems*, Cygnus Solutions, Inc., 1998

16. *GNUPro Tools for Embedded Systems*, Cygnus Solutions, Inc., 1999

17. *GNU Online Documentation*, Brian J. Fox, Free Software Foundation, 1994

18. *GNUPro Libraries*, Cygnus Solutions, 1998

19. *GNU Make*, Richard Stallman & Roland McGrath, Free Software Foundation, 1995

20. *Learning the Bash Shell*, Cameron Newham & Bill Rosenblatt, O'Reilly and Associates, Inc, 1998

21. *Learning GNU Emacs*, Debra Cameron & Bill Rosenblatt, O'Reilly and Associates, Inc, 1991

22. *Man-Pages*, Free Software Foundation, Inc., 2001

23. *Porting Unix Software*, Greg Lehey, O'Reilly and Associates, Inc, 1995

24. *Programming with GNU Software*, Mike Loukides & Andy Oram, O'Reilly and Associates, Inc, 1997, ISBN 1-56592-112-7

25. *The C Preprocessor*, Richard Stallman, Free Software Foundation, 1995

26. *The GNU C Library Reference Manual Volume One/Volume Two*, Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, & Ulrich Drepper Free Software Foundation, 1999

27. *Unix in a Nutshell*, Daniel Gilly and the Staff of O'Reilly and Associates, Inc, O'Reilly Associates, Inc, 1992

28. *Using and Porting GNU CC*, Version 2.8, Richard M. Stallman, Free Software Foundation, 1998

29. *Using and Porting GNU CC*, Version 2.95, Richard M. Stallman, Free Software Foundation, 1999

30. *Using GASP*, Roland Pesch, Free Software Foundation, 1996

31. *VIDE User Guide*, Dr. Bruce Wampler, 2001

32. *XScale Getting Started Guide*, Red Hat, 2001

33. Free Software Foundation Web Site (http://www.gnu.org), 2004

34. Macraigor mpDemon User Manual, 2004

35. Abatron BDI2000 User Manual, 2004

36. *ARM System Developer's Guide*, Andrew Sloss, Dominic Symes, and Chris Wright, Morgan Kaufmann, 2004

Glossary of Embedded System Terminology

TERM	DESCRIPTION
ABI	Application Binary Interface, which defines how programs should interface with the operating system, including specifications such as executable format, calling conventions, and chip-specific requirements.
API	Application Programming Interface, defining how programmers write source code that makes use of a library's or operating system's facilities by accessing the behavior and state of classes and objects.
BSP	Board Support Package, typically referring to the low-level code or scripts that build programs running on a particular CPU on a particular circuit board. Also refers to the ROM that boots an RTOS onto a specific board. Exact meaning varies.
Build	The process of configuring, compiling, and linking a set of tools. Also used as a noun, to denote the results of the process.
COFF	Common Object File Format. This debug format appeared with Unix SVR3, formerly common for Unix, and still used by some embedded systems. The Microsoft PE format for Windows is based on COFF.
COFF debugging	The debug format that is defined as part of the COFF specification.
Compiler	A tool that translates high-level source code in a language such as C or Pascal into machine-executable programs. The term may also refer specifically to the tool that translates from source to assembly language.
Debugger	A tool that allows programmers to examine and control a program, typically for the purpose of finding errors in the program.
Debug format	The layout of debugging information within an object file format. Debug formats include stabs, COFF, DWARF, and DWARF 2.
Debug protocol	The mechanism by which a debugger examines and controls the program being debugged.
DWARF	A debugging format based on attribute records. Versions include DWARF 1, 1.1, 2, & extensions to 2.
ECOFF	Extended COFF, a format used with MIPS & Alpha processors, both for workstations & embedded uses.
ELF	Extended Linker Format. Appeared with Unix SVR4 and used on many systems, including Solaris/SunOS, Irix, and Linux. Many embedded systems also use ELF – becoming very popular.
EL/IX	A set of configurable API's, based on a suitable subset of POSIX.1 and ISO C99, together with some extensions from Linux/GNU, BSD and SYSV, that are applicable to embedded applications.
Exception handling	Event that occurs when a block of code reacts to a specific type of exception. If the exception is for an error from which the tool, the debugger for instance, can recover, the debugger resumes its process.
Executable file	A binary-format file containing machine instructions in a ready-to-run form.
gas	Acronym for the GNU assembler. Interchangeably used with

TERM	DESCRIPTION
	capitalization, as GAS.
GDB	Main debugger used with GNU (command-line interface) The purpose of a debugger such as GDB is to allow you to see what is going on inside another program while it executes-or what another program was doing at the moment it crashed. GDB can do four main kinds of things to help you catch bugs in an embedded system: 1. Start your program, specifying anything that might affect its behavior 2. Make your program stop on specified conditions 3. Examine what has happened, when your program has stopped 4. Change things in your program, so you can experiment with correcting the effects of one bug and go on to learn about another You can use GDB to debug programs written in C and C++. GDB standard remote protocol: An existing ROM monitor used as a GDB backend.
GDBstub	Process and interfacing software to implement a protocol which is used for communication between the GDB debugger running on the host machine and the GDB debugger running on the target machine. In general terms, the scheme looks as follows: 1. On the "host": GDB already understands how to use this protocol; when everything else is set up, you can simply use the `target remote' command. 2. On the "target": You must link with your program a few special-purpose subroutines that implement the GDB remote serial protocol. The file containing these subroutines is called a "debugging stub".
GDBserver	This is a control program for Unix-like systems, which allows you to connect your program with a remote GDB via `target remote' - but without linking in the usual debugging stub. The GDBserver is not a complete replacement for the debugging stubs, as it requires essentially the same operating-system facilities that GDB itself does. In fact, a system that can run GDBserver to connect to a remote GDB could also run GDB locally!
GCC	Acronym for the GNU Compiler Collection – directly controls the C compiler. Interchangeably used with capitalization, as gcc.
gcj	Front end to GCC that is able to read Java `.class' files, generating assembly code.
Glibc	A Standard compliant library that has been ported to a number of operating systems, and provides ANSI/ISO, POSIX, BSD and System V compatibility.
GNU	Recursive acronym for GNU's Not Unix. A project to build a free operating system, started by Richard Stallman in 1985, with many useful spinoffs, such as the Emacs text editor, a C compiler, a debugger, and many other programming tools.

TERM	DESCRIPTION
GUI	Graphical User Interface, which refers to an interface and the techniques involved in using a keyboard or a mouse, for instance, to provide an easy-to-use interface to some software.
HAL	Hardware Abstraction Layer, which provides a portability layer to the operating system so that higher layers do not need to be aware of the specifics of the architecture and platform. This layer is designed to be comparatively small and simple to implement, and is also component-orientated to allow sharing between related platforms.
i386	Name for the 32-bit members of the Intel x86 family. Members include 386, 486, Pentium ("i586"), and Pentium Pro ("i686").
IDE	Integrated Development Environment, a GUI tool or a set of tools that uses GUI functionality.
ISA	Instruction Set Architecture.
Java™	An object-oriented, "write once, run anywhere" programming language, developed by Sun Microsystems.
JDK™	A software development environment for writing applets and applications in the Java programming language, developed by Sun Microsystems.
JIT	Just-in-time compiler that converts all of the byte code into native machine code just as a Java program is run, resulting in run-time speed improvements over code interpreted by a Java Virtual Machine (JVM).
JTAG	Joint Test Advisory Group, referring to a type of hardware interface that allows the testing of chips and boards within a complete system; programs running on processors with JTAG support may be controlled through the processor's JTAG port.
JVM	Java Virtual Machine, part of the Java Runtime Environment responsible for interpreting Java byte codes.
ld	The GNU linker. Interchangeably used with capitalization, as LD. See linker.
libgloss	The library for GNU Low-level OS Support, contains the startup code, the I/O support for GCC and newlib (the C library), and the target board support packages to which you need to port the GNU tools for an embedded execution target.
Linker	A tool that merges object files and library archives (such as compiled classes), building an executable, a complete program or a single executable file.
Linux	A free Unix operating system for many kinds of computers, created by Linus Torvalds and friends starting about 1990 (the pronunciation of /"lee-nuhks"/ is preferred, accenting the first syllable, since the name Linus has an /ee/ sound in Swedish).
Newlib	The Cygnus libraries, which include the C library, libc, and the math library, libm.
object file	A binary-format file containing machine instructions and possibly symbolic relocation information. Typically produced by an assembler.
object file format	The layout of object files and executable files. Common formats include a.out (or `b.out', for Intel 960 targets only)., COFF, & ELF
OSF/1	The Open Software Foundation's version of Unix, used in Digital's Alpha machines.

TERM	DESCRIPTION
Patch	A change in source code to correct or enhance processes.
PE	Portable Executable, Microsoft's object file format for Windows 95 and NT operating systems. It is basically COFF with additional header information.
PROM	Programmable Read-Only Memory, ROM that can be programmed using special equipment. PROMs can be programmed only once. Compare with EPROM.
RAM	Random-Access Memory, referring to volatile memory that can be read and written by a microprocessor.
RDI	Remote debugging library, used by ARM.
RDP	Remote Debugging Protocol, a protocol used with ARM's Demon monitor.
Registers	Registers are settings representing values that serve as temporary storage devices in a processor, allowing for faster access to data. Registers are divided into several classes: pseudo registers, temporary registers, and machine registers.
RISC	Reduced Instruction Set Computer, machines typically having fixed-length instructions, limited addressing modes, many registers, and visible pipelines. Examples include MIPS, ARM, SH, PowerPC and StrongARM.
ROM	Read-Only Memory, non-volatile memory that can be read, but not written, by the microprocessor.
RTOS	Real-Time Operating System.
Solaris	Sun's current version of Unix, superseding SunOS. Based on SVR4 Unix. Sun officially calls it SunOS 5.x, with versions including 2.0-2.6 (or, as Sun refers to them, 5.0-5.6).
Sparc	Name for the family of RISC processors based on Sun's SPARC architecture. Members include SPARClite, SPARClet, UltraSPARC, v7, v8, v9.
stabs	Based on symbol tables, a debug format originally introduced with the Berkeley Unix system, which records debugging information in certain symbols in the object file's symbol table. stabs information may also be encapsulated in COFF or ELF files.
stub	A small piece of code that executes on the target and communicates with the debugger, acting as its agent, collecting registers, setting memory values, etc. Also, in a native shared library system, the part of the shared library that actually gets linked with a program.
Target	The computer for which the compiler generates code. Used both to refer to an actual physical device, and to the class of devices.
TCP/IP	Transmission Control Protocol based on IP. This is an internet protocol that provides for the reliable delivery of streams of data across the web.
telnet	Standard terminal emulation protocol in the TCP/IP protocol stack, used for remote terminal connection, enabling users to log in to remote systems, thereby using resources as if connected to a local system.
Toolsuite	Informal term for the collection of programs that make up a complete set of cross-compilation tools. Typically consists of the following example's sequence: compiler->assembler->archiver->linker->debugger.

TERM	DESCRIPTION
Unix	Unix operating system. The uppercase spelling of `Unix' is used interchangeably. Invented in 1969 by Ken Thompson after Bell Labs left the Multics project, Unix subsequently underwent mutations and expansions at the hands of many different people, resulting in a uniquely flexible and developer-friendly environment. By 1991, Unix had become the most widely used multi-user general-purpose operating system in the world.
VFS	Virtual File System architecture.
Virtual Machine (VM)	An abstract specification for a computing device that can be implemented in different ways, in software or hardware. Compiling to the instruction set of a virtual machine is much like compiling to the instruction set of a microprocessor, using a byte code instruction set, a set of registers, a stack, a garbage-collected heap, and an area for storing methods.
X	An allegedly "over-sized, over-featured, over-engineered and incredibly over-complicated" window system developed at MIT and widely used on Unix systems. With its sources freely available, it is a vehicle that is widespread since developers can modify and customize it according to their requirements.
x86	Name for the Intel 8086 architecture family.
XCOFF	eXtended COFF , IBM's object file format for RS/6000 and PowerPC systems.

Index

A

Abatron BDI2000 · 101
ABI · 127
Acknowledgements · 7
addr2line · 46
-ahls · 58
-aln · 58
-ansi · 52
API · 127
ar · 46, 63
ARM Toolsuite · 115
as · 46
assembler · 43
assembly · 45
Assembly · 45
Automatic Display · 70

B

-b format · 61
bash · 113
Bash Shell · 14, 125
bc · 113
Bench++ · 8
Bibliography · 125
binutils · 113
bison · 113
Breakpoint Information Balloon · 80
Breakpoints and Watchpoints · 68
Breakpoints Window · 90, 97
BSP · 127
Build · 127
byacc · 113
bzip2 · 113

C

-C · 52
C language options · 51
C++ · 7, 45, 46, 51, 128
Changing Local Variables Values · 99
Code Display Drop-Down List Box · 83
COFF · 127
Command Line gdb · 65
compilation · 45
Compilation Options · 51
compiler · 43
Compiler · 127
Console Window · 92
Control Programs · 44
Controlling gdb · 72
Controlling Linker · 50
cpio · 113
cpp · 46
CPU Clock Rate · 9
Creating/Updating Libraries · 62, 63
crt0 · 62
cvs · 113
Cygwin · 112, 113, 125
Cygwin Key Features · 112
Cygwin Related Problems · 14

D

Debug format · 127
Debug protocol · 127
Debugger · 127
debugging · 45, 54, 56, 60, 117, 127, 128, 130
Deleting Breakpoints · 80
diff · 46
diff3 · 46
diffutils · 113
DOS style · 15
DWARF · 127

E

-e name · 61
ECOFF · 127
Editing Local Variables · 89
Elements of a Variable Structure · 90
ELF · 127
Environment Variables · 15
Exception handling · 127
Executable file · 127
Executing your Program · 68
Execution Control · 69
expect · 113
Expressions · 70

F

-ffast-math · 55
file · 113
File Drop-Down List Box · 82
fileutils · 113
findutils · 113
-finline-functions · 55
flex · 113
-fno-inline · 55

fortune · 113
Free Disk Space · 9
Free Software Foundation · 7, 126
FSF · 7, 52, 56, 57
ftp · 113
Function Browser Window · 93, 94
Function Drop-Down Combo Box · 82
-funroll-loops · 55

G

-g · 54
g++ · 46
gas · 113, 127
gasp · 46
gawk · 113
gcc · 46, 113, 128
gcj · 128
gdb · 46, 113, 128
gdb debugging session · 65
gdb Quick Reference · 67
gdb Scripts · 71
gdb under GNU Emacs · 73
gdbserver · 128
gdbstub · 128
gdbtk · 46
General Public License · 7
Getting Started · 11
Glibc · 128
Global Preferences Dialog · 77
Glossary of Embedded System Terminology · 127
GNU · 128
GNU Libraries · 64
GNU X-Tools Command Line Tool · 40
gnuchess · 113
GPL · 7
gprof · 113
grep · 113
groff · 113
GUI · 129
gzip · 113

H

Horizontal Scroll Bar · 82

I

i386 · 129
IDE · 129
inetutils · 113
Invoking GNU X-Tools · 41
Invoking ld · 59
ISA · 129
itcl · 113
itk · 113

J

Java™ · 129
JDK™ · 129
JIT · 129
JTAG · 129
JVM · 129

K

ksh · 113

L

ld · 46, 114, 129
-Ldir · 60
less · 114
libcurses · 114
libgloss · 46, 129
libjpeg · 114
libm · 46
libstdc++ · 46
Libstdc++ · 7
libtiff · 114
License Agreement · 7
linker · 43
Linker · 129
Linker Options · 60
Linker Scripts · 61
Link-Order Optimization · 62
Linux · 9, 11, 12, 127, 129
-lname · 60
Local Variables window · 98
Local Variables Window · 88

M

-M · 52, 61
M4 · 114
m68k-coff · 40
make · 46, 114
Make Related Problems · 15
man · 113, 114
Memory Preferences Dialog · 86
Memory Window · 85
Menus and Display Features · 81

Minimum System RAM · 9
Modes of gcc · 50
more · 114
Mount · 13
Multiple Threads · 99

N

-n · 61
Naming Conventions · 8
ncurses · 114
newlib · 7, 46
Newlib · 129
nm · 46
-nostartfiles · 52
-nostdlib · 53

O

-o name.x · 60
-O0 · 55
-O1 · 55
-O2 · 55
objcopy · 46
objdump · 46
object file · 129
object file format · 129
-oformat format · 61
Optimization · 55
Options to specify libraries · 52
OSF/1 · 129

P

-p · 54
Passing Options to the Assembler or Linker · 56
patch · 46
Patch · 130
PE · 130
-pedantic · 52
perl · 114
-pg · 54
POSIX · 64, 113, 127, 128
preprocessing · 45
preprocessor · 43
Preprocessor options · 52
Profiling Options · 54
Program Stack · 69
PROM · 130

R

-r · 61
RAM · 130
ranlib · 46, 63
rc file · 13
RDI · 130
RDP · 130
readelf · 46
Registers · 130
Registers Window · 84
registry entries · 14
Remove Carriage Returns · 15
Required CFLAGS / LFLAGS · 64
RISC · 130
ROM · 130
root · 11, 13, 41
RTOS · 130
run · 46
Run button · 75

S

-s · 60
sdiff · 46
Search Text Box · 83
sed · 114
Selecting a Breakpoint · 91
Selecting a Variabl · 89
Setting Breakpoints · 80, 96
-shared · 53
Shell Commands · 68
Signals · 71
simulator · 117
size · 46
Solaris · 127, 130
Source Browser · 94
Source File Selection · 95
Source Files · 73
Source Preferences Dialog · 76
Source Window · 75, 79
Sparc · 130
stabs · 130
Stack Window · 84
Starting gdb · 68
-static · 53
Status Text Box · 82
Stop button · 76
Stopping gdb · 68
strings · 46
strip · 46
stub · 128, 130
Sun · 129, 130
Symbol Table · 71

System Requirements · 9

T

tar · 114
tarball · 13, 40
Target · 130
Targets · 8
tcl · 114
TCP/IP · 130
telnet · 114, 130
Temporary Breakpoints · 91
texinfo · 114
textutils · 114
Threads · 94
tix · 114
tk · 114
toolchain · 13, 40, 41, 116
Toolchain Flow Diagram · 44
toolchains · 40, 41
Toolsuite · 130
Toolsuite Components · 8
-traditional · 52

U

Unix · 131
UNIX · 9, 50, 57, 63, 64, 125, 126, 131
unzip · 114
Using the GNU Assembler · 56
Using the Linker · 59

V

verbose · 51
VFS · 131
vi · 114
VIDE · 126
Viewing Local Variables · 96
vim · 114
Virtual Machine · 131
Visual gdb · 74
Visual gdb with External Debug Agents · 100

W

-w · 51
-W · 51
-Wa, option-list · 56
-Wall · 51
Watch Expressions Window · 86
Watch Menu · 87
-Werror · 51
which · 114
Window for Expressions · 81
Windows 2000 · 9
Windows 95 · 130
Windows NT · 9, 125
-Wl, option-list · 56
Working Files · 72
-Wtraditional · 51

X

-x · 60
x86 · 131
XCOFF · 131
Xemacs · 114

Z

zip · 114